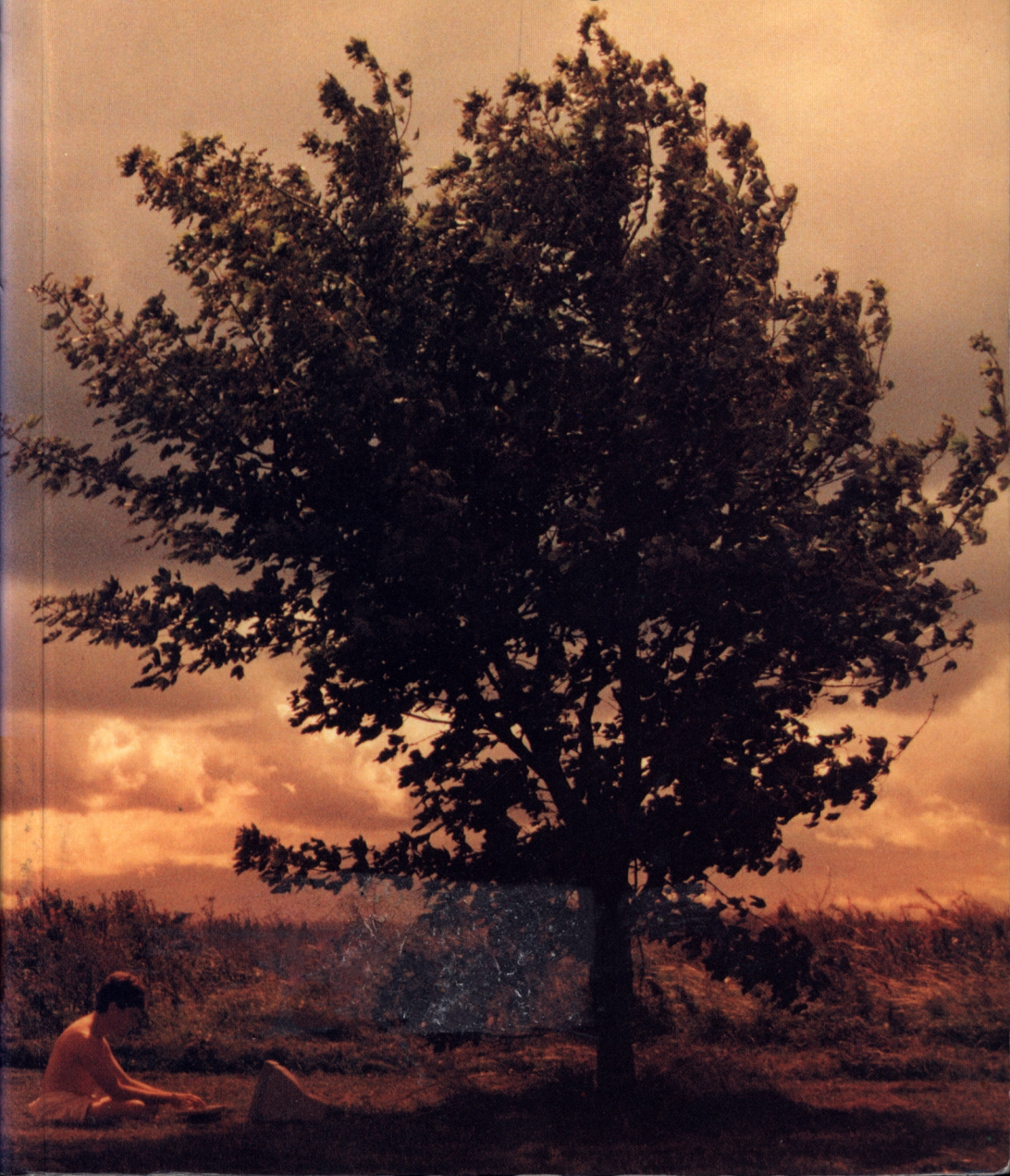


THE OS-9 GURU

1 - The Facts

Paul S. Dayan



THE OS-9 GURU

1 - The Facts

Paul S. Dayan

a *Galactic* publication

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

First Edition

ISBN 0 9519228 0 7

Copyright © 1992 by Paul S. Dayan

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, without the prior written permission of the author.

This book is sold as is, without warranty of any kind, expressed or implied. Neither the author nor the publisher shall be liable for any loss or damage caused or alleged to be caused directly or indirectly by this book.

Trademarks

OS-9 is a trademark of Microware Systems Corp. and Motorola Inc.

OS-9/68000 is a trademark of Microware Systems Corp.

UNIX is a trademark of AT&T Bell Laboratories

Microsoft and MS-DOS are trademarks of Microsoft Corp.

Compact Disc and Compact Disc Interactive are trademarks of Philips.

CP/M is a trademark of Digital Research.

This book was written for OS-9 version 2.4 (equivalent to CD-RTOS), but also describes changes and improvements to the operating system from version 2.2 onwards.

Published by:

Galactic Industrial Limited

Unit 3B

Mountjoy Research Centre

Stockton Road

Durham

DH1 3UR

United Kingdom



Fax (within the UK): 091 384774

Fax (outside the UK): +44 91 38477422

ERRATA

The following corrections should be made to the text of **The OS-9 Guru: 1 - The Facts** (first edition).

Page	Present text	Corrected text
fly	Fax (within the UK): 091 384774 Fax (outside the UK): +44 91 38477422	Fax (within the UK): 091 3847742 Fax (outside the UK): +44 91 3847742
147	main() {	main() { intercept(sighandler);
147	if (got_10) printf(.....	if (got_10) { got_10=FALSE; printf(..... }
147	else if (got_20) printf(.....	if (got_20) { got_20=FALSE; printf(..... }
147	else exit(....	if (invalid) exit(....
153	the process is returned the event value at the time the signal is received, and can	the process can
176	Only the creator of an alarm (same process ID)	Only the creator of an alarm (same process ID in OS-9 version 2.2, but same group number and user ID in later versions)

188, 189	The sharing of time slices can be calculated same proportion of time as that process.	There is no mathematical expression to calculate the share of processor time each process will get. It is an interesting corollary of the scheduling algorithm that all processes in a group with consecutive priorities will receive the same share of processor time.
270	1 even 3 odd	1 odd 3 even

This book is for Eliot, Ginette, Dinah, and Chloe.

About the author

Paul Dayan is a recognized world-wide authority on the OS-9 operating system. Born in London in 1956, he grew up in South Shields, in the North-East of England. After graduating from Cambridge University with an engineering degree, he worked briefly for ITT as an electronic engineer before joining Vivaway Microprocessor Consultants as technical director. Shortly afterwards Vivaway became Microware's UK distributor, and for the next six years Paul worked on OS-9 implementations and systems software, porting OS-9 to more systems than he can recall. When Microware set up their UK office in 1988, Paul left Vivaway and set up Galactic Industrial, to provide all forms of OS-9 consultancy. Having promised for years to write a book about OS-9, he finally got round to starting it in 1991. This is it - the result, he says, of having been marinated in OS-9 for ten years. Paul is married to Dinah Roy, a family doctor. They have a daughter, two dogs, and a cat.

About the company

Galactic Industrial was founded in 1988 by Paul Dayan. The company has three staff, and offers all forms of OS-9 consultancy throughout Europe, including systems analysis, real-time applications programming, CD-I programming, systems programming (device drivers and file managers), OS-9 implementations, on-site trouble-shooting, and on-site training courses. Galactic Industrial also sells a range of OS-9 development tools, file managers, and device drivers.

Acknowledgements

I would like to thank Tony Mountifield, Steve Weller, and Ole Hansen who kindly proofread this book. Their invaluable suggestions greatly reduced the number of errors, and increased the readability of the book. I would also like to thank Nick Rainey who read the book and allowed me to print his opinion on the back cover.

Preface

OS-9 was written over a decade ago, at a time when the IBM PC did not exist, and CP/M was considered the last word in operating systems for microcomputers. Quietly but relentlessly it has grown to become one of the most popular operating systems world-wide, particularly in industrial applications and home computers. Yet, despite the fact that OS-9 is now a venerable grandfather in the time scale of microcomputer software, it is surging ahead even more strongly, and – with the advent of Compact Disc Interactive – is set to throw off its veil of shyness and take the software world by storm.

This amazing longevity in the most rapidly changing market of all is a tribute to the foresight and imagination of the creators of OS-9 – Ken Kaplan, Larry Crane, and Bob Doggett. They designed into OS-9 innovative features that gave it a broader applicability and a longer technical life than any other operating system. The result is an operating system that has achieved widespread popularity despite not being backed by any high volume manufacturer, a feat only matched by UNIX, which had the advantage of being widely used in universities before being commercially launched.

Although OS-9 is very widely used, it does not yet have a high press profile – the worlds of industry and home computing don't seem to excite the press in the same way that business computing does. So to date very few books have been written about OS-9, and many users have had the uneasy feeling that they are not getting the best out of this powerful programming environment.

This book – the first of a series covering all aspects of OS-9 – was written to fill this gap. It is aimed at a wide audience, from novice computer users, through experienced applications programmers, up to systems programmers seeking to adapt or extend OS-9. This is not a "chatty" book. It is intended as a readable reference work, to give the reader the facts about OS-9. Nor does it repeat information that is readily available from the Microware OS-9 User's and Technical Manuals. This book gives the inside information about

PREFACE

how the operating system works, in complete and authoritative detail. And it describes how the features of the operating system should be used in applications programs.

The aim of the book – to dispel the mystique of OS-9; to blow away the grey fog that clouds the programmer's view of the operating system, and give him the information and the confidence to use this powerful tool to the full.

Paul S. Dayan

Author's note: human beings are both male and female, but the English language does not have a common pronoun to cover both. So, for brevity, masculine pronouns have been used throughout this book, but are to be read as both masculine and feminine. PSD

About this book

The OS-9 Guru is aimed at a broad spectrum of readers, from naïve computer users, through applications programmers, to systems programmers. It forms an introduction and technical reference for the OS-9 operating system. The OS-9 Guru is not intended as a replacement for the Microware OS-9 manuals. Indeed, throughout the book it is assumed that you also have the OS-9 User's Manual, the OS-9 Technical Manual, and the OS-9 C Compiler Manual, so information readily available from these manuals is not repeated here. Instead, **The OS-9 Guru** aims to clarify all of the important aspects of the use of OS-9, and to give every technical detail about the operating system that an applications or systems programmer might find useful.

Chapters 1 to 5 are an introduction to OS-9 (and to operating systems in general), and contain information that will be useful to any user of OS-9.

Chapters 6 to 11 contain more detailed information about how OS-9 works, and describe the facilities of OS-9 that are available to the applications programmer, including multi-tasking and inter-process communications.

Chapters 12 to 14 are intended primarily for systems programmers, and describe the detailed internal workings of OS-9, with particular emphasis on the I/O system. However, these chapters will also be of interest to advanced applications programmers.

Chapter 15 covers the special features of the Microware C compiler, including how to use C with assembly language, and how to write operating system components in C. This chapter also contains some useful tips for all C programmers.

The Glossary at the end of the book defines some common computing terms, and their particular meaning in the context of the OS-9 operating system.

PREFACE

Conventions used in this book

I have tried to keep to a constant format throughout the book, particularly with regard to technical terms. The important conventions are described below:

hexadecimal numbers	Prefixed by a '\$' character. For example: \$4AFC.
file names and pathlists	Enclosed in single quotes. For example: '/dd/startup'.
command line options	Enclosed in single quotes. For example: '-rv=disk'.
keys	Shown in brackets, such as [CR] for the carriage return (ENTER or RETURN) key. [^A] refers to Control-A – that is, hold down [CTRL] and press [A].
utility names	In bold text. For example: dir .
module names	In bold text. For example: init .
program symbols	In bold text. For example: D_Proc .
error numbers	Shown either using the assembly language symbols defined in the file '/dd/DEFS/funcs.a', for example E\$BusErr , or the C language symbols defined in the file '/dd/DEFS/errno.h', for example E_BUSERR .
error handling in examples	For clarity, many of the example code fragments omit the handling of errors. In practice, error handling statements should be used with any statement that could generate an error.

There is also a brief glossary at the end of the book explaining many of the technical terms used.

CHAPTER 1. OS-9 - THE OPERATING SYSTEM	1
1.1. WHAT IS AN OPERATING SYSTEM?	1
1.2. WHY IS AN OPERATING SYSTEM IMPORTANT?	2
1.3. WHAT IS MULTI-TASKING?	3
1.4. PROGRAMS IN ROM	4
1.5. THE FUNCTIONS OF AN OPERATING SYSTEM	4
1.6. A COMPARISON OF OPERATING SYSTEMS	6
1.7. THE MAIN PROPERTIES OF OS-9	6
1.7.1. Multi-tasking	6
1.7.2. Real Time	6
1.7.3. ROMmable	8
1.7.4. Modular	9
1.7.5. Unified Device Independent I/O System	9
1.7.6. Inter-process Communication Functions	9
1.7.7. High Performance	10
1.7.8. Adaptation to New Hardware	10
1.7.9. Complete Set of Functions	10
1.7.10. Broad Spectrum of Applications	11
1.7.11. The Future	11
1.8. THE PARTS OF OS-9	12
1.8.1. Utilities	12
1.8.2. Language compilers	12
1.8.3. Kernel	12
1.8.4. File managers	12
1.8.5. Device drivers	13
1.8.6. Device descriptors	13
1.8.7. Program support modules (trap handlers)	13
1.8.8. Customization modules	14
1.9. SPECIAL FEATURES OF OS-9	14
CHAPTER 2. USING OS-9	17
2.1. BOOTING OS-9	17

CONTENTS

2.2. SHELL - THE COMMAND LINE INTERPRETER	21
2.3. SHELL BUILT-IN COMMANDS	24
2.4. ENVIRONMENT VARIABLES	25
2.5. PATHLISTS	26
2.6. CURRENT DIRECTORIES	27
2.7. INPUT LINE EDITING	28
2.8. OTHER SPECIAL KEYS	29
2.9. MULTIPLE PROCESSES ACCESSING THE TERMINAL	30
2.10. A TYPICAL DIRECTORY STRUCTURE	30
CHAPTER 3. OS-9 MODULES, MEMORY, AND PROCESSES	33
3.1. THE OS-9 MEMORY MODULE	33
3.2. A PROGRAM MODULE	35
3.2.1. Sync Word	35
3.2.2. System Revision ID	36
3.2.3. Module Size	37
3.2.4. User and Group	37
3.2.5. Offset to Module Name	37
3.2.6. Access Permissions	37
3.2.7. Type and Language	39
3.2.8. Attributes and Revision	40
3.2.9. Edition Number	43
3.2.10. Other Fields	43
3.2.11. Header Parity	43
3.2.12. Offset to Program Entry Point	44
3.2.13. Offset to Default Trap Entry Point	44
3.2.14. Minimum Program Data Space	45
3.2.15. Minimum Program Stack Size	45
3.2.16. Offset to Data Initialization Table	45
3.2.17. Offset to Data Pointers Initialization Table	46
3.2.18. Module CRC	47
3.3. MODULES IN FILES	48

3.3.1. Module Groups	49
3.4. OS-9 MEMORY	50
3.4.1. Coloured Memory	51
3.4.2. Memory Allocation	53
3.4.3. Inter-task Memory Protection	55
3.5. PROCESSES AND MULTI-TASKING	57
3.5.1. A Dead Process	60
3.5.2. System State Processes	60
CHAPTER 4. THE OS-9 UTILITIES	63
4.1. WHAT IS A UTILITY?	63
4.2. UTILITY SYNTAX	64
4.2.1. Formal Syntax Notation	65
4.3. UTILITIES FOR OPERATING SYSTEM FUNCTIONS	66
4.4. SYSTEM MANAGEMENT UTILITIES	68
4.5. GENERAL UTILITIES	68
CHAPTER 5. SYSTEM MANAGEMENT	71
5.1. THE SYSTEM MANAGER	71
5.2. THE FILING SYSTEM	72
5.3. THE PASSWORD FILE	72
5.4. SYSTEM STARTUP	73
5.5. THE .LOGIN FILE	74
5.6. DISK FORMATTING	74
5.7. INSTALLING A BOOT FILE	75
5.8. ARCHIVING	78
CHAPTER 6. C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER	81
6.1. THE DEVELOPMENT SYSTEM	81
6.2. THE C COMPILER	82
6.3. FILE NAMING CONVENTIONS	83
6.4. CC OPTIONS	84
6.5. THE ASSEMBLER	87

CONTENTS

6.5.1. The psect Directive	88
6.5.2. The vsect Directive	89
6.5.3. External Symbols	90
6.6. THE LINKER	91
6.6.1. Linker Options	92
6.7. THE PROGRAM DEBUGGER	93
CHAPTER 7. THE OS-9 I/O SYSTEM	97
7.1. I/O SUB-SYSTEMS AND DEVICES	98
7.2. FILE MANAGERS AND DEVICE DRIVERS	102
7.3. DEVICE DESCRIPTORS	103
7.4. PATHS, PATHLISTS, AND FILES	105
7.5. PERMISSIONS, ATTRIBUTES, AND MODES	108
7.6. THE I/O SYSTEM CALLS	110
7.6.1. I\$Attach: Add Device to Device Table	111
7.6.2. I\$Detach: Remove Device from Device Table	113
7.6.3. I\$Dup: Duplicate a Path	113
7.6.4. I\$Create: Create a File	114
7.6.5. I\$Open: Open a Path	115
7.6.6. I\$MakDir: Create a New Directory	116
7.6.7. I\$ChgDir: Change Current Directory	117
7.6.8. I\$Delete: Delete a File	117
7.6.9. I\$Seek: Set the File Pointer	117
7.6.10. I\$Read: Read Data	118
7.6.11. I\$Write: Write Data	118
7.6.12. I\$ReadLn: Read Line	118
7.6.13. I\$WritLn: Write Line	118
7.6.14. I\$GetStt: Get Status	119
7.6.15. I\$SetStt: Set Status	120
7.6.16. I\$Close: Close a Path	120
7.6.17. I\$SGetSt: Get Status on System Path	120
7.7. PATH DESCRIPTOR OPTIONS	121
7.7.1. RBF Options Section	123

7.7.2. SCF Options Section	125
7.7.3. SBF Options Sections	128
7.8. MAKING A NEW DEVICE DESCRIPTOR	129
7.9. SPECIAL FEATURES	134
7.9.1. RBF Disk Caching	134
7.9.2. SCSI Device Driver System	135
7.9.3. Ethernet support	136
7.9.4. The X Window System	136
CHAPTER 8. INTER-PROCESS COMMUNICATION	137
8.1. WHY USE MULTI-TASKING?	137
8.2. WHAT IS INTER-PROCESS COMMUNICATION?	138
8.3. OS-9 INTER-PROCESS COMMUNICATION FACILITIES	139
8.4. FORKING A PROCESS	140
8.5. SIGNALS	142
8.5.1. Masking Signals	145
8.5.2. Signals - Cautions	149
8.6. EVENTS	149
8.6.1. Using Events	154
8.6.2. Pulsing an event	154
8.6.3. Interlocked handshake	155
8.6.4. Buffered handshake	155
8.6.5. One to many synchronization	156
8.6.6. Rendezvous	158
8.6.7. Semaphore	158
8.7. PIPES	159
8.7.1. Using Unnamed Pipes	161
8.7.2. Using Named Pipes	163
8.8. DISK FILES	164
8.8.1. RAM Disks	166
8.9. DATA MODULES	169
8.10. SHARED EXTERNAL MEMORY	173
8.11. ALARMS	175

8.11.1. System State Alarms	178
CHAPTER 9. MULTI-TASKING	183
9.1. OS-9 PROCESS SCHEDULING	183
9.2. THE SCHEDULER FEATURES	185
9.3. ACTIVATING A PROCESS	186
9.4. AUTOMATIC SCHEDULING	188
9.5. AN EXAMPLE OF SCHEDULING	189
9.6. SCHEDULING PRE-EMPTION MECHANISMS	190
9.6.1. Minimum Priority	190
9.6.2. Maximum Age	192
9.6.3. Seizing Control	193
9.6.4. The Precedence of the Mechanisms	194
9.7. SCHEDULING IN REAL TIME APPLICATIONS	194
CHAPTER 10. EXCEPTION HANDLING	197
10.1. EXCEPTION HANDLING UNDER OS-9	198
10.2. USER AND SYSTEM STATE RETURN	199
10.3. SYSTEM CALLS - TRAP #0	200
10.4. TRAP HANDLER MODULES - TRAPS #1 TO #15	202
10.4.1. The Trap Handler Routine	204
10.4.2. Installing Trap Handlers	206
10.4.3. Terminating Trap Handlers	208
10.4.4. Writing a Trap Handler in C	208
10.5. HARDWARE EXCEPTIONS	212
10.5.1. Hardware Exceptions in User State	213
10.5.2. Example - Bus Error Handler	217
10.5.3. 'move from sr' and 'move from ccr'	219
10.5.4. Hardware Exceptions in System State	220
10.6. INTERRUPTS	222
10.6.1. How 68000 Interrupts Work	222
10.6.2. Using Interrupts Under OS-9	223
10.6.3. Interrupts OS-9 Cannot Handle	227

10.6.4. The Level 7 Interrupt	228
10.7. THE EXCEPTION VECTOR TABLE	229
10.8. THE EXCEPTION JUMP TABLE	230
CHAPTER 11. OS-9 SYSTEM CALLS	233
11.1. THE SYSTEM CALL MECHANISM	233
11.2. SYSTEM CALL PARAMETERS	234
11.3. CUSTOM SYSTEM CALLS	235
11.4. USER AND SYSTEM STATE CALLS	236
11.5. THE SYSTEM CALLS	236
11.5.1. F\$AllTsk System Call	242
11.5.2. F\$CCtl System Call	243
11.5.3. F\$ChkMem System Call	245
11.5.4. F\$DatMod System Call	245
11.5.5. F\$DelTsk System Call	246
11.5.6. F\$FModul System Call	246
11.5.7. F\$GBlkMp System Call	247
11.5.8. F\$GProcP System Call	248
11.5.9. F\$Gregor System Call	248
11.5.10. F\$GSPUMp System Call	249
11.5.11. F\$IODel System Call	250
11.5.12. F\$Load System Call	250
11.5.13. F\$Permit System Call	251
11.5.14. F\$Protect System Call	253
11.5.15. F\$SysDbg System Call	254
11.5.16. F\$SysID System Call	255
CHAPTER 12. DEVICE DRIVERS	257
12.1. THE FUNCTION OF A DEVICE DRIVER	257
12.2. DEVICE STATIC STORAGE	258
12.3. PATH DESCRIPTOR	263
12.3.1. RBF Path Descriptor	263
12.3.2. SCF Path Descriptor	268

CONTENTS

12.4. SYMBOLIC DEFINITIONS	271
12.5. REGISTER USAGE	272
12.6. DEVICE DRIVER ROUTINES	273
12.6.1. Initialize	274
12.6.2. Terminate	276
12.6.3. Read	277
12.6.4. Write	281
12.6.5. Get Status and Set Status	284
12.7. INTERRUPTS	293
12.7.1. Solicited Interrupts	298
12.7.2. Unsolicited Interrupts	301
12.7.3. Choosing Interrupt Levels	304
12.8. A SKELETON DEVICE DRIVER	305
12.9. CLOCK DRIVERS	308
CHAPTER 13. FILE MANAGERS	313
13.1. THE FUNCTION OF A FILE MANAGER	313
13.2. FILE MANAGER ROUTINES	315
13.3. KERNEL ACCESS TO THE FILE MANAGER	316
13.4. PARAMETER CONVENTION	317
13.5. PATHLISTS	318
13.6. CREATE AND OPEN	318
13.6.1. SCF	318
13.6.2. RBF	319
13.6.3. The File Descriptor Sector	320
13.6.4. The Allocation Bit Map	322
13.6.5. Access to the Whole Disk	324
13.7. CHANGE DIRECTORY	324
13.8. MAKE DIRECTORY	325
13.9. DELETE	327
13.10. SEEK	328
13.11. READ AND WRITE	328

13.11.1. RBF	328
13.11.2. SCF	329
13.12. READ LINE AND WRITE LINE	329
13.12.1. RBF	330
13.12.2. SCF	330
13.13. GET STATUS AND SET STATUS	331
13.14. CLOSE	333
13.15. CALLING THE DEVICE DRIVER	333
13.16. RESOURCE CONTROL	334
13.17. A SKELETON FILE MANAGER	336
CHAPTER 14. OS-9 INTERNAL STRUCTURE	341
14.1. THE SYSTEM GLOBALS	342
14.2. THE OTHER SYSTEM MEMORY STRUCTURES	343
14.2.1. Process Descriptor	345
14.2.2. Path Descriptor	345
14.2.3. Module Directory	347
14.2.4. Device Table	348
14.2.5. Device Static Storage	348
14.2.6. Process Descriptor Table	350
14.2.7. Path Descriptor Table	350
14.2.8. Interrupt Polling Table	351
14.2.9. Event Table	351
14.2.10. Service Dispatch Tables	352
14.3. SYSTEM GLOBALS STRUCTURE	353
14.4. PROCESS DESCRIPTOR STRUCTURE	361
CHAPTER 15. MICROWARE C AND ASSEMBLY LANGUAGE	373
15.1. MICROWARE C	373
15.2. CIO AND MATH TRAP HANDLERS	374
15.3. THE REMOTE DIRECTIVE	374
15.4. PROGRAM STARTUP	376

CONTENTS

15.5. C WITH ASSEMBLY LANGUAGE	376
15.6. REGISTER VARIABLES	377
15.7. CODING FOR SPEED	378
15.8. THE 'LINK' INSTRUCTION	380
15.9. A FUNCTION IN ASSEMBLY LANGUAGE	381
15.10. STRUCTURE RETURN	382
15.11. CALLING C FROM ASSEMBLY LANGUAGE	384
15.12. A DEVICE DRIVER IN C	387
15.13. A FILE MANAGER IN C	396
15.14. HINTS ON C PROGRAMMING	403
15.14.1. Declarations and Definitions	403
15.14.2. Pointers and Arrays	404
15.14.3. Pointers and Functions	406
15.14.4. Pointers and Structures	408
APPENDIX A GLOSSARY	411
APPENDIX B SBF DEFINITIONS	417

CHAPTER 1

OS-9 - THE OPERATING SYSTEM

1.1 WHAT IS AN OPERATING SYSTEM?



OS-9 is an operating system. The name is an abbreviation of "Operating System for the 6809 microprocessor". OS-9 was originally written by Microware under contract to Motorola, who wanted to demonstrate that the 6809 microprocessor was sufficiently powerful to be the heart of a true computer. OS-9/6809 is therefore owned jointly by Microware and Motorola. OS-9/6809 was very popular in its own right, and was used on a number of different high-volume home and educational computers. Microware later rewrote OS-9 for the 68000 family of microprocessors, and it is in this form that OS-9 has reached its current very wide popularity on the industrial market. But what is an operating system?

Depending on your viewpoint as a user, you might see it as one or more of the following:

- The kernel of software that gets the computer going.
- The user interface and command interpreter.
- A provider of a basic I/O interface for the user.
- A set of functions for applications programmers.
- A core of standards and documentation for applications programmers.
- A set of philosophies for programmers.

An operating system will (usually) provide all of these things. But these are features as seen from the point of view of a user. To appreciate what an

operating system does we must look at it from a system designer's viewpoint. A computer consists of electronic hardware – central processing unit (CPU – the processor), memory, disk drives, displays, keyboards, printers, and so on. The user wants to run **applications programs** to carry out specific tasks, such as a word processor, spreadsheet, or language compiler. Somewhere there must be software that controls and manages these "hardware resources". This software *could* be in the application program. However, that has very important disadvantages:

- The software must be present in every program.
- Programs are not "portable" to computers with different hardware.
- The computer cannot run multiple programs concurrently, as they will clash in their use of hardware resources.

An alternative approach is to use an operating system. The operating system is a body of software that provides functions to allocate and manage the hardware resources. It divides up the processor time between multiple programs running concurrently, allocates memory to programs as they need it, and arbitrates between programs trying to use the same input/output (I/O) device.

In addition, the operating system provides software to implement commonly required functions, such as disk file management. This provides standards for programmers, and significantly reduces the work required to write applications programs.

The operating system is not a program in itself. It provides functions for use by programs. But the operating system vendor will usually also provide a set of commonly required programs for general system maintenance, such as file copying, deleting, and display. These programs are known as "utilities".

1.2 WHY IS AN OPERATING SYSTEM IMPORTANT?

Many companies have in the past elected to write their own operating systems, in the belief that this gives them greater control over the functionality of the operating system, and a fuller understanding of all the functions.

In reality, however, real time kernels, executives and operating systems written in house "re-invent the wheel" (creating techniques and software already available for purchase), take a great deal of support and documentation, and are vulnerable to the departure of key personnel.

The learning curve for this type of project is usually very long, unless programmers with particular "systems programming" experience are hired. An operating system requires very different programming techniques and philosophies from those of applications programming.

There are other advantages to using a bought-in, widely used operating system. A known, documented, and fixed environment simplifies large projects, and new generations of applications. Additional functionality to simplify the applications programming is likely to be available, because a generally accepted operating system available on a wide range of hardware promotes the development of third-party software products and documentation. It also gives long-term confidence to manufacturers and users.

1.3 WHAT IS MULTI-TASKING?

A multi-tasking operating system is one which provides for the concurrent running of multiple programs. As the processor of a computer can only run one program at a time, multi-tasking is achieved by running one program for a short time (perhaps 20 milliseconds), stopping it (saving its state so it can be restarted later), starting the next program, and so on. Provided the time interval - known as a "time slice" - is short enough (whether it is will depend on the application), this gives the illusion of running the programs concurrently.

There are two (very different) uses of a computer for which multi-tasking is very important. The first is a multi-user system. This is a computer system which has multiple users working on it (on different terminals!) at the same time. Each user is running his own program independently of the others - the application programs do not interact with each other.

The second use is a multi-tasking application. Here multiple programs work together to implement a single application. This is how most computer-based industrial and domestic products work. For example, one program handles the operator interface, while a second program collects data from sensors, and a third program communicates with a central factory computer. This multi-tasking approach is very important. It allows each program to concentrate on a single job, without having to "poll" (check) frequently to determine whether other jobs need doing - they will be done concurrently by the other programs.

In a multi-tasking application the programs must work together to perform the overall job. This requires the passing of data between the programs. It

also requires synchronization between the programs. For example, one program must not continue its job until another program has collected data for it. These functions of synchronization and data transfer are known as inter-process communication. Correct use of these functions is essential to the writing and working of a multi-tasking application.

1.4 PROGRAMS IN ROM

Computer-based products vary widely in their complexity and cost. A product may have a powerful processor, with disk drives, high-quality displays, and a networking connection to other computers. Or, it may be a low cost, "embedded" product, with no operator interface or disk drives.

If the system has no disk drives (or other suitable storage medium), the operating system and all of the programs must reside in ROM. Therefore the capability for the operating system to be ROMmed (placed in ROM), and to support programs in ROM, is essential to low-end applications.

Traditionally this group of applications has been serviced by "real time kernels" (not to be confused with the OS-9 "kernel" module!) or "executives". These differ from an operating system in that the programs and real time kernel are all linked together to make one software package, which is then typically ROMmed. In general, new programs are not dynamically loaded and run. Also, traditionally a real time kernel does not automatically share out the time of the processor between the multiple programs of the application. Instead, programs are assigned a priority, and the highest priority program runs until it asks to be suspended.

This approach produces a final software package that is (in general) somewhat smaller, and (because the real time kernel offers less functionality) perhaps a little faster. However, development and debugging are more difficult, and more complex applications cannot easily be implemented. So a ROMmable operating system is attractive, allowing one programming environment to be used for a very wide range of applications.

1.5 THE FUNCTIONS OF AN OPERATING SYSTEM

In the light of all the above "desirable features", we might therefore hope that an operating system would provide the following functionality (the terms used are explained in later chapters):

☐ **Management of system resources**

- Memory
- processor time
- I/O devices
- Co-processors
- External events
- Inter-process communication mechanisms
- Inter-user protections

☐ **Provision of commonly required functionality**

- Disk file handling
- Input line editing
- Program loading
- Date and time
- Terminal functions
- Process debugging
- I/O device control
- Multi-user management
- Utilities

☐ **Application program portability (hardware independence)**

- Complete set of system functions (so the programmer never needs to access hardware directly)
- Standardized error numbering
- Device-independent I/O functions

☐ **Ease of system customization**

- Modular operating system structure, especially I/O
- ROMmable operating system and programs
- Isolation of areas requiring customization
- Ease of new system build
- Independence of filing systems from hardware functions

□ **Standardization of software and documentation**

- Well-defined operating system functions
- Modular approach to system customization and extension

1.6 A COMPARISON OF OPERATING SYSTEMS

Having drawn up a long wish-list of functions for a hypothetical operating system, it is interesting to compare different well-known operating systems for their functionality – see figure 1 on the next page.

1.7 THE MAIN PROPERTIES OF OS-9

The list of features in figure 1 makes OS-9 appear very attractive. This is perhaps not so surprising. OS-9 was not written in a hurry. It was developed over a period of about three years by a small group of programmers at Microware under contract to Motorola (who therefore jointly own OS-9/6809). The programmers carefully considered existing operating systems (particularly UNIX), building on the good features of existing technology, but inventing new techniques wherever they saw the need.

This makes OS-9 one of the very few operating systems developed through a long-term project as a commercial product outside of a hardware manufacturer. As a result OS-9 implements almost all of the functions that might be wished of an operating system, in an elegant, straightforward way. Indeed, it is a good model for the academic study of a broad-spectrum, multi-tasking, real time operating system. This section summarizes the main properties of OS-9, with a brief description of how each is implemented.

1.7.1 Multi-tasking

Using a hardware timer (whatever is available on the particular system) to generate "tick" interrupts, OS-9 performs automatic time-slicing between any number of programs. In addition, OS-9 has several mechanisms to allow the advanced programmer to modify this "scheduling", even to change it to the non-automatic priority-only scheduling expected in a real time kernel.

1.7.2 Real Time

The term "real time" is often abused or misunderstood. It means that real world events are being processed as they happen. A real time system is one that must respond to an external event within a specified time. For example,

	OS-9	UNIX	MS-DOS
Multi-tasking	Yes	Yes	No
Multi-user	Yes	Yes	No
Real time	Yes	No	No
Modular	Yes	No	No
Broad spectrum applicability	Yes	No	No
Large systems	Yes	Yes	No
Single-user workstations	Yes	Yes	Yes
Personal computers	Yes	No	Yes
Home computers (diskless)	Yes	No	No
Embedded industrial products	Yes	No	No
Processors available	Motorola Intel	Many	Intel
Memory limitation	None	None	640k
Requires Memory Management Unit?	No	Yes	No
ROMmable (and can operate diskless)	Yes	No	No
Device-independent I/O system	Yes	Yes	No
User-customizable I/O device drivers	Yes	No	Yes
User-customizable I/O file management	Yes	No	No
User-customizable kernel	Yes	No	No
Dynamically customizable	Yes	No	No
Robust disk filing structure	Yes	No	No
All versions generally compatible	Yes	No	No
Associated with a hardware manufacturer	No	Yes	Yes
Accepted by major manufacturers	Yes	Yes	Yes
Wide third-party software/documentation	No	Yes	Yes
Next generation standard	Yes	Yes	No

• Figure 1 - A comparison of operating systems

an image processing system must process the image of one part before the next part on the conveyor belt comes along. If it cannot, then it has failed its function. Therefore the real time response time will vary from application to application, and between external events in the same application. It could be as much as five minutes, or as little as 500 nanoseconds.

A non-real-time system may also be specified to respond within a certain time, but it is not fatal if it does not. For example, an accounts computer may be specified to respond to an operator input within two seconds, but if it takes three seconds on occasion that is only an annoyance to the operator.

Computers respond to external events by polling or by interrupts. Polling - repeated testing for a condition - is relatively slow and uncertain - a program may take a while to perform a particular task before it is ready to poll again. An interrupt is a hardware signal to the processor. It causes the processor to suspend execution of the current program, and execute a separate function (called an "interrupt handler"). The interrupt handler carries out any tasks that require "immediate" attention, and/or sends a software signal to the program that wants to know about the external event. This would typically wake up the program, which would be in a suspended state waiting for the event.

Response by interrupt is therefore very much more efficient and reliable than response by polling, although it requires somewhat more initial learning and programming. A real time operating system must offer a way of adding new interrupt handlers, and mechanisms for interrupt handlers to communicate with programs. OS-9 has both. Also, interrupts are not very useful without multi-tasking, as the single program could not sleep, waiting for the interrupt - it would still need to poll to see if the interrupt had occurred. Therefore a real time operating system must also be multi-tasking (which OS-9 is).

1.7.3 ROMmable

The unique memory module mechanism of OS-9 (described later) makes the operating system and application programs inherently ROMmable. Also, there is no need to provide any information as to where in ROM the modules are - at startup the OS-9 kernel scans all the ROM areas to find all modules present in ROM, and builds a module directory indicating where they are. So modules can be placed in ROM in any order, and with gaps between them if desired. Once entered in the module directory, a module is located by its name, in a similar way to named files on a disk.

1.7.4 Modular

The operating system itself is separated into several memory modules. This allows very easy customization. If certain functionality (such as disk filing) is not required, that module is simply omitted. If additional functionality is required – even dynamically at run-time – additional modules (such as the Internet Support Package for Ethernet networking) can be loaded, and their functionality is immediately available.

1.7.5 Unified Device Independent I/O System

A device independent input-output (I/O) system is one in which the same basic functions – such as "open", "read", and "write" – are used by a program to access all types of I/O devices. This simplifies programming, and permits "redirection" – a program written to write to a terminal can have its output sent to a disk file without being modified in any way.

An operating system with a unified I/O system is designed to allow the I/O system to be easily customized, extended, or reduced. All I/O calls from programs go to the operating system kernel, the core functionality of the operating system which is present in all configurations. The kernel provides a common environment to manage the call, and passes it on to the appropriate subroutine within the operating system.

OS-9 has a particularly well structured I/O system. I/O calls go to the kernel, which then calls a file manager appropriate to the class of device (disk drive, tape drive, terminal, network, and so on). The file manager has the job of handling the logical data manipulation, such as the file handling on a disk. The file manager does not know how to access the physical device, and calls a device driver specific to that device whenever physical I/O is required (for example, reading and writing of sectors on the disk).

The kernel, file managers, and device drivers are all separate memory modules, so new device drivers can easily be added for new physical devices, and new file managers can be written for new classes of device.

1.7.6 Inter-process Communication Functions

OS-9 has several different inter-process communication functions available for use by application programs. They are discussed in detail in the chapter on "Inter-process Communication", as they are very important to the effective generation of multi-tasking applications.

1.7.7 High Performance

OS-9 was designed with execution speed and code size very much in mind, and so was written in assembly language. This makes it very well suited to even small, cost-sensitive products.

However, because OS-9 is written in 68000 family assembly language, it cannot be adapted ("ported") to other processor families. The greatly increased power of microprocessors, and reduced cost of memory, since OS-9 was first written has reduced the need for the operating system to be as small and fast as possible. Therefore Microware has written a companion operating system - OS-9000 - in C, which can readily be adapted to different microprocessors. In particular, OS-9000 is available for most 80386 and 80486 IBM PC compatible computers.

1.7.8 Adaptation to New Hardware

The modular arrangement of the I/O system and other parts of the operating system makes OS-9 very adaptable to new hardware, without the need for any of the source code of hardware-independent parts, such as the kernel.

The user can also customize the operating system, by adding or removing memory modules. This applies to all parts of the operating system - even the kernel can be adapted or extended, using "kernel customization modules".

1.7.9 Complete Set of Functions

It is very important that the programmer can always work within the operating system environment. He must not need to bypass the operating system and access hardware directly because of limitations of the operating system, as this reduces portability of the application, and can destroy the multi-tasking capability. OS-9 provides a full set of functions for the management and allocation of all resources, but it is not unnecessarily complex. Therefore the programmer can learn how to work within the operating system environment without too much effort (so he is much more likely to do so!).

Although OS-9 can be reduced for low-end applications, all of its functions are sophisticated enough to support top-end applications as well. For example, the disk file manager provides a full hierarchical directory structure, with almost unlimited file size, long file names (28 characters), and true record locking. Microware has additional file managers for a wide range of applications, such as graphics and networking, all equally sophisticated.

1.7.10 Broad Spectrum of Applications

OS-9 is suitable for a broader range of applications than perhaps any other operating system available. This comes from its modular, ROMmable construction, its full set of sophisticated functions (including multi-user support), and its relatively small size. OS-9 can be (and has been) used in small, diskless, embedded products, on large multi-user systems, and on everything in between, such as personal computers and home computers.

Microware is a wholly private company, and owes no allegiance to any hardware manufacturer. This makes OS-9 unusual, as most other popular operating systems are partly or wholly owned by a hardware manufacturer. So Microware's only aim is to develop and support OS-9 as a commercially attractive operating system. It cannot be coerced into making OS-9 less accessible, or into bending it to be more suited to particular uses and less suitable to others. And it cannot be shut down at the commercial whim of a parent company.

This gives much greater confidence to companies who commit to using OS-9. The operating system is the environment for the whole product. It is much easier to change hardware than to change operating system!

Perhaps surprisingly, given its relatively low market profile, OS-9 is one of the most popular and widely used operating systems in the world, and has sold many hundreds of thousands of copies. It has been used in high volume products - such as Fujitsu and Tandy home computers - and in high-tech products - such as the Space Shuttle control station.

1.7.11 The Future

OS-9 is already the most widely used real time operating system on Motorola microprocessors. The advent of Compact Disc Interactive (CD-I) may well make it the most popular operating system in all fields. CD-I - launched at the end of 1991 in the USA and Japan, and early in 1992 in Europe - is a standard developed by Philips in conjunction with Sony to use compact disc for the storage of audio, video, and computer data, as well as programs. The operating system in the player is OS-9 (under the name CD-RTOS), and all CD-I applications run under OS-9.

Philips, Sony, and most of the other major Japanese domestic electronics manufacturers are or will be producing CD-I players, and are hoping that CD-I will be as big as, or bigger, than audio compact disc. If so, every field will be open to OS-9, which could conceivably become the standard

operating system in industrial products, home computers, personal computers, and workstations.

1.8 THE PARTS OF OS-9

OS-9 is a sophisticated operating system made up of several parts, and often supplied with accessory programs. This section summarizes these software components. Microware, and third-party software vendors, also supply a wide range of complementary software products.

1.8.1 Utilities

Microware license OS-9 in a number of guises, containing more or less of the operating system components and companion products. The programmer is most likely to be using Professional OS-9, which comes with a large set of utility programs, a C compiler, and the **umacs** screen editor. These are not part of the operating system itself, which is an environment for programs to run under. They are the basic programs that every programmer is likely to need in order to develop his software. Microware also sell additional utility programs for particular purposes, such as the C Source Level Debugger.

1.8.2 Language compilers

Professional OS-9 comes with a C compiler. Microware also have Pascal, Fortran, and Basic compilers. Third-party software suppliers offer compilers for other languages, such as Modula-2.

1.8.3 Kernel

The kernel module is the core of the operating system. It contains all of the system-independent functions, such as multi-tasking support and memory allocation. The kernel also provides the environment for all I/O calls to be serviced by the I/O system (which is composed of the file managers and device drivers).

1.8.4 File managers

The file manager modules perform the logical data processing part of an I/O call. For example, the maintenance of the filing structure on a disk, or input line editing on a terminal. In general, each file manager is suitable for a class of devices, such as disk drives, tape drives, or a network. The file managers

do not know how to physically transfer data through an I/O interface. For this they make calls to the device drivers. OS-9 can support any number of file managers, and each file manager can work through any number of device drivers. This gives OS-9 a "tree structured" I/O system.

1.8.5 Device drivers

Each device driver module has the functions to control a particular I/O interface device, such as a disk controller chip, or a serial communications chip. The device driver does not know why it is requested to perform the I/O, and (in general) performs no data manipulation or interpretation - that is the job of the file manager that is calling the device driver.

This functional split between file managers and device drivers simplifies new implementations and customizations of OS-9. To add a new type of I/O interface which fits an existing class it is only necessary to write a new device driver. The device driver writer does not need to have any understanding of how the filing system works.

1.8.6 Device descriptors

Somehow the operating system must know what device a program is referring to, and must be able to select the appropriate file manager and device driver to manage the I/O request. OS-9 does this using memory modules known as device descriptors. These modules contain only data. They give the name of the device (equal to the name of the device descriptor module), the name of the file manager module, the name of the device driver module, and information about the hardware configuration (such as the memory address of the interface chip).

1.8.7 Program support modules (trap handlers)

Programs can access memory modules by name. A program makes a call to the operating system, passing the name of the desired module, and the operating system (having searched the module directory) returns the memory address of the module. Modules can therefore be used to store data, or sets of subroutines for use by programs.

OS-9 provides another way by which a program can call subroutines in a separate module, without having to determine the address of the module. These modules are known as trap handlers. The calling program informs the operating system of the names of the trap handler modules it wishes to use.

Then, when it wants to call one of the subroutines in the trap handler, it makes another operating system call, specifying the trap handler and the number of the function within the trap handler.

1.8.8 Customization modules

OS-9 is very customizable. It is even possible to customize the kernel. The user supplies an additional module containing functions to add to or replace the existing system calls. The name of the additional module (or modules - any number are permitted) is placed in the configuration data module **init**. On startup, the kernel finds all such modules, and calls their initialization functions. This allows the module to add new system calls or replace existing ones, so extending or modifying the functionality of the kernel.

1.9 SPECIAL FEATURES OF OS-9

OS-9 is an unusual operating system in many respects. It uses advanced techniques to allow it to address a very wide spectrum of applications effectively, and yet remain small. OS-9 also makes *no* demands regarding the hardware configuration it runs on. It can run programs with nothing more than a processor and some memory (although without some I/O a system cannot do anything useful!).

☐ OS-9 modules

The OS-9 memory module concept is central to much of the flexibility of OS-9. It allows the operating system to be dynamically configurable, for programs and operating system components to be in ROM, and for programs to be held in memory even when not running. Programs can also use memory modules as common data pools, for inter-process communication.

☐ Relocatable, re-entrant, ROMmable operating system and programs

Microware specifies that all programs (and operating system components) must be written relocatably (or "position independent"). This means that the program does not use any absolute program addresses. Instead, program accesses such as calls to subroutines are done relative to the current program counter, so the program module can be placed anywhere in memory without needing modification.

The 6809 and 68000 family processors are specifically designed to support this mode of programming, so it is not a restriction. As a result, OS-9 does not need any memory management hardware to translate memory addresses (this is done in other operating systems so that the program always logically lives at the address for which it was written).

Microware also specifies that programs must access their data memory "register indirect". That is, when a program is started the operating system sets one of the processor's registers to point to an area of memory that the operating system has allocated for the data memory of the program. The program then accesses its variables relative to this register. This allows programs to live in ROM, while having variables in RAM. It also permits multiple "incarnations" of the program to run concurrently. For example, several users can be using the **umacs** editor at the same time. Only one copy of the program exists in memory, but OS-9 allocates separate data memory for each incarnation (or "process"), so each functions independently of the others.

This type of program (where data accesses are all register indirect) is known as a re-entrant program. Again, the Motorola processors are designed to support this kind of addressing.

The language compilers all automatically generate position-independent, re-entrant code.

☐ **Tree-structured I/O system**

The OS-9 I/O system is separated into file managers, device drivers, and device descriptors. This fragmentation of the I/O system into completely separate modules makes the OS-9 I/O system very customizable.

☐ **Dynamically modifiable I/O system**

OS-9 is very unusual - its I/O system is dynamically modifiable while the system is running. New file managers, device drivers, and device descriptors can be loaded at any time. An I/O interface can be used in one way, and then used for a completely different purpose by loading a new file manager and/or device driver. A manufacturer of an I/O board can supply a device driver with it that can be loaded whenever the user wishes to use the board.

Not only does this add to the flexibility of the system, but it makes debugging new I/O system components very much easier, as the system does not have to be rebooted to test each revision.

☐ **Customization hooks throughout**

OS-9 is intended for a very wide range of applications, many of which will have unique requirements. Microware have therefore made almost every aspect of OS-9 customizable - often in more than one way - by providing well defined mechanisms to modify or extend the operating system.

☐ **User interface is very similar to UNIX**

Microware modelled the user view of OS-9 very much on UNIX. Users with experience of UNIX are therefore rapidly at home with OS-9 - although there are differences!

The programmer's view of OS-9 is also very similar to UNIX, especially at the C programming level. Most UNIX programs can be ported (at the source code level) to OS-9 with little or no modification.

☐ **Very regular utility command line syntax**

The utilities provided by Microware all conform closely to the same command line syntax. Options are preceded by a '-', and can be in any order anywhere on the command line. All utilities respond to the '-?' ("help") option, and common options are the same on different utilities - for example, '-z' is used to indicate that file names will come from standard input.

This significantly improves the user-friendliness of the operating system, and encourages third-party software suppliers to use the same syntax conventions.

CHAPTER 2

USING OS-9

2.1 BOOTING OS-9



In most computers the main memory (Random Access Memory - RAM) loses its data when the power is turned off. At power-on or reset the CPU cannot assume that it has any valid programs in RAM. Therefore every computer has a small amount of ROM (Read Only Memory) - memory that cannot be written, but does not lose its contents on power-off.

The ROM contains a "bootstrap" program that (in general) reads a known part of a disk to load the operating system into RAM. The bootstrap program then jumps to the coldstart routine of the operating system, which does the rest of the startup procedure. This operation is known as "bootstrapping", from the analogy of someone lifting themselves up by their own bootlaces - how can the operating system start up the computer if the operating system itself is not in memory?

The term "bootstrapping" is usually abbreviated to "booting", and "bootstrap program" to "boot program".

The boot program is not itself part of the operating system, nor can it use any of the facilities of the operating system. It is a self-contained program whose job is to load the operating system into the computer's memory.

Because the boot program must initialize and use the hardware of the particular computer it is running on, it is not provided as a compiled program by Microware. Instead, Microware gives the implementor example source code which the implementor must adapt. In consequence, the booting

procedure is not the same on all OS-9 computers, so the description here is necessarily a bit vague.

Microware supplies the implementor with a simple debugger program to go with the boot program, known as the ROM-based debugger. This, too, is a stand-alone program that does not use any of the operating system facilities. The implementor has the option of including this debugger with the boot program. Also, from OS-9 version 2.4, Microware supplies the implementor with a much improved debugger, known as ROMbug. The implementor has the option of including this with the boot program, in place of the old ROM-based debugger.

If one of these debuggers is included with the boot ROM, then normally the system will enter the debugger on power-on or reset. The user must then enter the "go" command:

```
debug: g[CR]
```

to continue with the booting procedure. (Note: [CR] means press the **RETURN** or **ENTER** key). On some systems, the implementor will have written the boot program to bypass entering the debugger if a switch or link on the CPU board is appropriately configured.

Prior to OS-9 version 2.4, the example source code provided by Microware could only boot from one disk drive, or (if assembled differently) search for the OS-9 kernel in ROM. As most systems want to boot from hard disk normally, but from floppy disk when needed, many implementors have modified the example source code to allow booting from multiple drives. Because this is not in the Microware-supplied code, it varies from one manufacturer to another. Some implementors also incorporated the search for the kernel in ROM. This type of boot program will typically search for the kernel in ROM first. If the kernel is found, the boot program jumps to the coldstart routine of the kernel in ROM. Otherwise, the boot program attempts to boot from each disk drive it knows about in turn, until it is successful with one of them.

From OS-9 version 2.4, Microware provides the implementor with an additional set of example source code known as "CBOOT" (because it is written in C, rather than assembly language). This has the capability to boot from multiple drives, including tape drives, and even across a network.

So, a typical boot-up sequence would be:

- Power-on or hit the reset button! A "booting" message may be displayed on the terminal.

- The ROM-based System Debugger may be present (giving a display of the processor's internal registers):
debug: **g**[CR]
- You may be prompted to select which drive you want to boot from.
- The boot program will now find the OS-9 kernel in ROM, or read the operating system boot file from the (selected) drive. If you had "enabled" the ROM-based debugger with the **e** command, the boot program will re-enter the debugger. The "go" command will continue the boot. This allows the implementor to check that the boot has worked correctly. The boot program then locates the kernel module, and jumps into its coldstart routine.
- The kernel searches the ROM and the boot file (if any) for modules, and enters them in the module directory. If you had enabled the ROM-based debugger, the kernel will re-enter the debugger. This allows the implementor to set breakpoints in any of the modules in the boot file – the kernel has already checked their CRCs, and will not check them again. The **g** command will continue the OS-9 coldstart.
- The kernel links to the configuration module **init** (to get the user-configurable initialization parameters), initializes all its tables and other memory structures, and opens the default input and output paths (usually the terminal '/term') and default directories (usually the hard disk root directory). It then starts (forks up) the program whose name is given in the **init** module – the initial program to fork. This program is usually called **sysgo**.
- The last operation the kernel does in its coldstart routine is to fork the "system process". This is a program whose code is contained within the kernel module. Its job is to manage the wakeup of programs that are in timed sleep, or have set alarms. This system process is not visible to the user.
- Having forked the system process, the kernel coldstart ends. The operating system executes no further code unless called by a program or an interrupt.
- The **sysgo** program provided by Microware first changes its execution directory to 'CMDSD' (on the drive whose name is given in the **init** module). It then forks up the **shell** program – the OS-9 command line interpreter – with its input directed to come from a file called 'startup', rather than from the keyboard. When that **shell** has finished processing the instructions in 'startup', **sysgo** enters a loop,

forking a **shell**, then waiting for it to die. Thus if the user terminates the **shell** program, **sysgo** will start up another one.

- The 'startup' file contains **shell** command lines to initialize the system. For example, it may call the **tsmon** utility to log in other terminals.
- If the computer hardware does not have a battery-backed calendar/clock chip, 'startup' will have a line to call the **setime** utility, and you will be prompted for the date and time:

```
yy/mm/dd hh:mm:ss [am/pm]
Time: 92/08/15/09/30
```

- Once **shell** has processed the 'startup' file, **sysgo** forks up a new shell, which presents you with its default prompt:
\$
- This ends the startup procedure.

Because the incarnation of **shell** that processes the 'startup' file is not the same incarnation as the one **sysgo** subsequently forks to give you the command line prompt, "private" changes made in instructions in the 'startup' file are not passed on to the **shell** that gives you the prompt. Examples of changes that are "private" to the executing program are changes to the default directories (**chx** and **chd**), and changes to the **shell** prompt string (**-p=...**).

Some users will modify the 'startup' file so that its last instruction is to login the system console (terminal). In this case the **shell** processing the 'startup' file will never finish (because the **tsmon** program used to log in the system console never finishes). The 'ex' built-in command of the **shell** can be used to transform the shell into an incarnation of **tsmon** as the last act of the 'startup' file, to avoid an unnecessary incarnation of shell. If **tsmon** has been called, you must press [CR], which will give you the **login** prompt, rather than the **shell** prompt.

The **shell** program "inherits" the current directories of the **sysgo** program . After bootup, the "current data directory" is the root directory of the "initial device", as specified in the **init** module (unless the 'startup' file calls **tsmon** to log in the system console, in which case the login procedure may change the current data and execution directories). Typical device names are:

```
/d0    floppy disk drive 0
/h0    hard disk
```

/dd "default device" - usually the hard disk

and the "current execution directory" is the 'CMDS' directory within that root directory:

```
/d0/CMDS    floppy disk drive 0
/h0/CMDS    hard disk
/dd/CMDS    "default device"
```

Notice that in OS-9 all device names start with a '/' character. A typical system might use the following device names:

```
/d0    floppy disk drive 0
/d1    floppy disk drive 1
/fh0    hard disk without format protection
/h0    hard disk drive
/h0fmt  same as '/fh0'
/mt0    tape drive
/nil    "null" device - data sent here is lost
/p      parallel port (Centronics) configured for use with a printer
/p1     second serial port configured for use with a printer
/p2     third serial port configured for use with a printer
/r0     RAM disk
/term   first serial port (system console)
/t1     second serial port
/t2     third serial port
/u0     floppy disk drive 0 configured for Microware Universal form
        if '/d0' is some other format
```

• **Figure 2 - Typical device names**

2.2 SHELL - THE COMMAND LINE INTERPRETER

shell is a program - it is not built into the operating system. No programs or utilities are built into the operating system itself. **shell** reads in a line (typically from the keyboard), and processes it. The line may contain:

- the name of a program to fork, with parameters:
\$ `dir CMDS -e`
- the name of a text file containing shell command lines:
\$ `my_proc_file`
- shell "built-in" commands:
\$ `chd /dd/USER`

shell accepts several special characters to modify its default behaviour – see figure 3.

Many **shells** may be running concurrently, for the same or different users. Each user has at least one **shell**. Because **shell** is just a program, a user can run a different command line interpreter, in place of **shell** (for example, **mshell**, the advanced command line interpreter from Microware). Also, **shell** may be called (forked) from within another program. For example, both **basic** and **debug** have a '\$' command to fork a **shell**, and the same can be achieved from **umacs** with the key sequence [[^]X][C] ([[^]X] means "hold [CTRL] and press [X]").

shell implements "wild carding" on file names, using any combination of the characters '*' (for "any number of characters or none") and '?' (for "any single character"). **shell** actually performs wild card comparison of names using the "compare names" system call (**F\$CmpNam**).

```
$ dir *.c
```

will display the names of all files that end in '.c'.

```
$ list fred?.c
```

will list all files whose names start with 'fred', followed by any single character, followed by '.c'.

Be aware that it is the **shell** that reads the current data directory to resolve wild-carded file names, not the program that is being forked. The program is passed the expanded file names as parameters just as if you had typed them in full.

Command line parameters are separated by spaces. If a parameter is to contain spaces or **shell** special characters, it must be enclosed in single or double quotes. For example:

```
$ echo *
```

will print the names of all files in the current data directory, while:

```
$ echo "*"

```

will print a single '*' character.

The present OS-9 **shell** does not have UNIX-like parameter substitution or flow control "language" features – it is a simple command line interpreter. Microware also sells a much more sophisticated command line interpreter – **mshell**.

- `;` separates commands to execute sequentially:
`$ dir -e ; mdir`
- `&` separates commands to run concurrently – **shell** does not wait for the child to finish before executing the next part of the command, or giving a new prompt if the `'&'` is at the end of the line:
`$ list fred & dir`
`$ tsmon /t1 &`
- `!` "pipes" the output of the first program into the input of the second program:
`$ echo fred ! list -z`
- `^` sets the priority of the program being forked:
`$ dir ^200`
- `#` sets the size of the data space of the program, in kilobytes. This modifier is rarely used – utilities dynamically allocate buffer memory themselves:
`$ basic #20k`
- `()` forks up a separate **shell** to execute the commands between the parentheses. The separate **shell** can change its private parameters (such as current data directory) without affecting those of the parent **shell**:
`$ (chd /dd/SYS ; list errmsg)`
- `<` redirects the standard input path of a program being forked to any device or file:
`$ list -z <file_list`
- `>` redirects the standard output path of a program being forked:
`$ list fred >/p`
- `>>` redirects the standard error path of a program being forked:
`$ dsave /d1 >>err_log`
 the redirection modifiers can be combined to redirect any two or all of the standard paths to another device or file:
`$ r68 fred.a -q1 >>>/p`
`$ shell <>>>/t1`

• Figure 3 – shell special characters

2.3 SHELL BUILT-IN COMMANDS

Certain desirable commands cannot be separate utility programs, as they change private properties of this "incarnation" of **shell**, or operate on "children" of the **shell** (programs forked by the **shell**). Programs forked up by **shell** receive a copy of its environment, but if they change their own environment this has no effect on the environment of the parent **shell**.

chd	change shell 's data directory: \$ chd /dd/USER
chx	change shell 's execution directory: \$ chx /dd/USER/ETC/CMDS
kill	kill another process (by process ID): \$ kill 7
w	wait for one (any) child of the shell to die: \$ w
wait	wait for all children of the shell to die: \$ wait
setenv	set (or change) an environment variable: \$ setenv TERM vt100
unsetenv	forget an environment variable: \$ unsetenv PATH
setpr	set execution priority of a process: \$ setpr 6 200
logout	exit this shell (same as "end-of-file" key): \$ logout
profile	execute the commands in a text file (does not fork another shell to process the text file - contrast just typing the file name): \$ profile fred
ex	"chain" rather than "fork" another program - the shell effectively dies after forking the program: \$ ex tsmon /term
-e	enable full error message printing
-ne	print only error numbers
-l	only allow exit via "logout", not "end-of-file" key
-nl	allow exit via "logout" or "end-of-file" key
-p	enable display of prompt

<code>-p=<str></code>	set new prompt string: \$ <code>-p="George: "</code>
<code>-np</code>	disable display of prompt
<code>-t</code>	echo input lines – useful for procedure files
<code>-nt</code>	do not echo input lines
<code>-v</code>	display directory searching
<code>-nv</code>	do not display directory searching
<code>-x</code>	abort on error (shell program terminates) – the default if processing a procedure file
<code>-nx</code>	do not abort on error – the default if taking input from the keyboard (SCF or GFM device).

2.4 ENVIRONMENT VARIABLES

When a process is forked by the **shell** (or by the **os9exec()** C library function), it is passed a parameter string composed of two parts: the command line parameters, and the environment variables. Each environment variable is a character string with an associated name. For example, the environment variable **TERM** may be assigned the string **vt100**. The **shell** built-in commands **setenv** and **unsetenv** are used to create and delete environment variables local to that **shell**. Any name can be invented, and assigned any character string. When the **shell** forks a process, it passes a copy of all the environment variable names and strings that it currently has. If that process then forks another process itself, it will pass on a copy of the same environment variables (provided it uses the **os9exec()** C library function to do the fork), unless it has changed its copy of the environment variables.

It is important to bear in mind that the environment variables do not exist globally in the system. Each process has, in its static storage memory, a copy of the environment variables passed to it when it was forked. The process can modify, delete, or add to them, and pass them on to any process it forks. This means that different processes can have the same environment variable name with a different character string.

The environment variables are effectively implicit command line parameters. They save you the trouble of typing in many additional parameters with each command line. A program can look through the list of environment variables it has been passed, to see if there are any names it recognizes. For example,

the **umacs** screen editor looks for the environment variable **TERM** to tell it which type of terminal is being used. The shell looks for the environment variable **PATH** to tell it what directories to search when forking a program, in addition to the current execution directory (which it searches first), and the environment variable **HOME** to tell it which directory to change to if the **chd** command is used without a pathlist.

The **printenv** utility is used to list all the currently defined environment variables of the process that forks it (usually your command line **shell**).

2.5 PATHLISTS

All I/O devices and files are accessed by means of pathlists. A pathlist is a text string identifying the device or file.

If the pathlist starts with a '/', the first name element is a device name:

```
/p
/term
/dd
```

otherwise the pathlist is relative to the current data directory or current execution directory (depending on whether the file is opened with "execute mode", which is a function of the program using the pathlist). For example, the **load** utility opens the file with the execute mode (unless the '-d' option is used), and so the pathlist is relative to the current execution directory:

```
$ load mdir
```

whereas the **list** utility opens the file without the execute mode, so the pathlist is relative to the current data directory:

```
$ list myfile
```

Note: a device name is the name of the device descriptor module describing the device.

The disk file manager of OS-9 ("RBF") supports hierarchical directories. Therefore a pathlist may have any number of name elements. Name elements are separated by '/':

```
/dd/CMDS/BOOTOBSJS/h0fmt
MYDIR/myfile
```

Letter case is not significant in device names (or any module names), or in **RBF** pathlists. By convention, directories are created with upper case names, to be easily visible in a directory listing:

```
$ mkdir NEWDIR
$ copy /dd/startup newdir/startup
```

Each directory contains an entry '.', referring to itself, and '..', referring to its parent. For example:

```
$ dir ..
```

displays a directory listing of the directory one level above this – the parent directory of the current data directory.

```
$ dir ../BROTHER
```

displays a directory listing of the directory 'BROTHER', which is a directory with the same parent as the current data directory – a "sibling" of the current directory.

OS-9 also permits multiple '.' to go further up the hierarchy. Thus '...' is equivalent to '../..' (both refer to three levels up). The '..' entry of the root directory refers to itself (it is identical to the '.' entry). If your pathlist has more '.' than there are levels to go up the directory hierarchy, this peculiarity of the root directory avoids any problems – the extra '.' are effectively discarded, as going to the parent of the root directory returns to the root directory.

Note: anything in the pathlist beyond the initial device name is a function of the file manager, not the kernel. For example, the '.' convention described above is a function of the disk file manager **RBF**.

2.6 CURRENT DIRECTORIES

There are separate current data and execution directories for each process (running program).

If a pathlist starts with a '/', the first pathlist name element is a device name, and the "current directory" feature is not invoked:

```
$ list /h0/SYS/password
```

The "current directory" feature is invoked if a pathlist does not start with a '/'. The current execution directory is used if the path is opened with the "execute" mode, otherwise the current data directory is used. For example, the **dir** utility opens the target directory without the execute mode (unless

the '-x' option is used), so a pathlist that does not start with '/' is relative to the current data directory:

```
$ dir USER/ROBERT
$ dir -x BOOTOBSJS
```

The **load** utility opens the file to load with the execute mode (unless the '-d' option is used), so the pathlist is relative to the current execution directory:

```
$ load BOOTOBSJS/h0fmt
$ load -d OBSJS/myprog
```

A child process inherits the current directories from the parent process. It can change them, but this does not affect the current directories of the parent (or of any other process). In the above examples, **dir** and **load** inherited the current directories of the **shell** that forked them.

The **shell**'s current directories may be changed using the built-in commands **chx** and **chd**:

```
$ chx /h0/CMDS
$ chd .../PROJECT/SOURCE
```

2.7 INPUT LINE EDITING

Line editing is a function of the I/O subsystem handling the input device - the file manager and device driver - not a function of **shell** (although **mshell** does its own line editing). For terminals, the "Sequential Character File manager" (SCF) is normally used. The same line editing is therefore available for most keyboard line entry.

SCF uses a line buffer for the editing, passing the finished line (when [CR] is pressed) to the calling program. The buffer is 512 bytes, so this is the maximum length of a line typed in, including the [CR]. A separate buffer is allocated for each path opened, so line input on one path does not affect another path.

The editing keys are all customizable (using the **tmode** and **xmode** utilities). Setting a key code to zero suspends the feature. The usual key assignments are as follows ([^X] means hold [CTRL] and press the [X] key):

<u>Key</u>	<u>Action</u>
[BS] or [BkSp]	delete one character left
[^X]	delete the whole line
[^D]	reprint the whole line (useful on teletypes!)

<u>Key</u>	<u>Action</u>
[^A]	redisplay line buffer from cursor to end-of-line character (useful for repeating a command, perhaps with editing)
[ESC]	end-of-file if the first character on the line

2.8 OTHER SPECIAL KEYS

Certain other input keys have special effects. These are a function of the device driver. As with the line editing keys, the utilities **xmode** and **tmode** can be used to change the special keys:

<u>Key</u>	<u>Action</u>
[^E]	abort - kill the last process to use the terminal
[^C]	interrupt - normally kills the last process to use the terminal

[^E] causes the device driver to send a "quit" signal to the last process that used the input device. [^C] causes the device driver to send an "interrupt" signal to the last process that used the input device. As with other signals, if the process has not installed a signal handler function (and most utilities do not), the kernel will terminate the process.

shell does install a signal handler, and so receives and handles the signals. If **shell** was the last process to use the terminal, [^E] causes it to kill the last child forked (by sending it the "quit" signal) and return to the prompt. [^C] causes it to return to the prompt without killing the child, effectively putting the child into the "background". So if the user enters a command line, and the program has done no input or output to the terminal, the user can decide to put the program into the "background" by pressing [^C], as if the command line had been terminated by the '&' character:

```
$ pr fred >/p
[^C]
$
```

If a process is waiting for an I/O operation to complete when it is killed, the I/O operation is not corrupted - the operating system completes the I/O operation before terminating the process. However, if the I/O operation is a read or write of a terminal or printer (through the **SCF** file manager), the operation is aborted, as **SCF** does not support a filing system that could be corrupted. Note that if a write operation is aborted in this way, any

characters that have already been written to the device driver's buffer, but are still waiting to be sent to the device, will be transmitted despite the abort, unless the device was about to be shut down in any case (no paths are open on the device). Therefore, unless special care is taken, it is difficult to know whether such characters will be transmitted after an abort (but this is not usually important).

The third special key causes the device driver to request **SCF** to pause output at the end of the next line – any other key restarts the output:

<u>Key</u>	<u>Action</u>
[^W]	pause output at end-of-line

2.9 MULTIPLE PROCESSES ACCESSING THE TERMINAL

SCF queues concurrent accesses to the terminal, whether for input or output. For example, if, when **shell** is waiting for a line of input, a background process attempts to write to the screen (perhaps to display an error message), **SCF** puts it to sleep. The error message will come out when the **shell**'s request is finished – an input line has been typed – and the background process is woken up by the operating system.

In particular, the **shell** built-in command 'w' causes **shell** to wait for a child process to die. This releases the terminal (because the **shell** is waiting for the process to die, rather than requesting keyboard input), and allows the background process to print its error message and die.

2.10 A TYPICAL DIRECTORY STRUCTURE

OS-9 is very customizable, and has been implemented on a wide range of hardware, so many things will vary from system to system. This includes the directory arrangement on the main disk drive. However, most implementors try to retain the example arrangement from Microware, the main elements of which are described here.

Most OS-9 systems will have a primary mass storage device, either a floppy disk or a hard disk. The first floppy disk drive is usually known as '/d0', while a hard disk is usually known as '/h0'.

An "alias" (in the form of an additional device descriptor) is usually provided for each of the devices, so that each may be known as '/dd' (default device). Only one **dd** device descriptor can be loaded at any one time. Usually this

will be the hard disk drive if the system has one, but some configurations use a "RAM disk" as the default device.

Many programs use '/dd' as the route to definitions files and other program-specific data. Therefore you should load a "dd" appropriate to the device you want used for such purposes. For example, a typical command to load the **dd** device descriptor for the hard disk is:

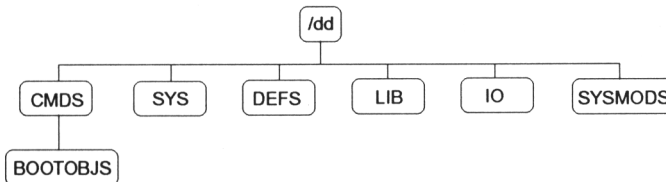
```
$ load BOOTOBSJ/dd.h0
```

Such a load command can be placed in the 'startup' file, although some implementors name '/dd' as the initial device in the **init** module. In such configurations the **dd** module must be in the boot file, or ROM, so to change the default device a new boot file or ROM must be made (or a **dd** device descriptor with a higher revision number can be loaded).

The root directory of this device will usually contain at least the following directories:

CMDS	utilities execution directory
DEFS	header (definitions) source files for assembly language and C programs
LIB	library files (for use by the linker)
SYS	system management text files
SYSMODS	system customization source files
IO	device descriptor (and device driver) source files

and 'CMDS' will also normally contain the directory 'BOOTOBSJ'. This directory contains all the OS-9 modules used to make up the operating system, which allows you to create customized boot files, plus any operating system modules not in the boot file, for loading as required.



• **Figure 4 – A typical directory structure**

CHAPTER 3

OS-9 MODULES, MEMORY, AND PROCESSES

3.1 THE OS-9 MEMORY MODULE



The OS-9 memory module concept is at the heart of many of the innovative features that make OS-9 applicable to such a wide range of applications. It permits the operating system to keep track of programs, operating system components, and common data areas in memory. Irrespective of the absolute location of these items in memory, programs can locate them by passing a name to the operating system, similar in some ways to accessing a file on a disk drive.

An OS-9 module is a program, data structure, operating system component, or any string of bytes, with an identifying header stuck on the front and a Cyclical Redundancy Check (CRC) tagged on the end.

The header contains information about the module, to prevent its unauthorized or incorrect use, and facilitate automatic mechanisms such as finding all modules in ROM at startup, initialization of the data space of a program, and protection against data corruption by module CRC checking. Very importantly, the header also contains an offset to the module name string.

A module is known by its name. All modules currently in memory must have different names (unless of a different type or language – see below), although modules with the same name may be contained in files (for example, on disk).

The addresses of all modules in memory are held in the "module directory", which is a table built and maintained in memory by the operating system. Each entry in the module directory contains the address of the module, the current number of "links" (uses) to the module, a group identifier (explained

later), and a module header parity check value, to guard against corruption of the module header.

When a program or operating system function wants to locate a module in memory, it makes an operating system "link to module" call (**F\$Link**), passing the name of the module. The kernel looks at each module directory entry in turn. The module address points to the module header, from which the kernel finds the name of the module, and compares this with the name passed to it. This is repeated until it finds the module requested.

[The term "link" causes some confusion, as it has many meanings in software terminology. In the OS-9 environment it has two principal meanings:

- a) To get the address of a module in memory, by using the operating system **F\$Link** system call.
- b) To build a program (or other module) from one or more **Relocatable Object Files** (ROFs), using the "linker" utility.

There is no connection at all between these two functions. The OS-9 linker utility is called **l68**.]

Modules may be present in ROM at startup, or be loaded from disk (or other I/O device) at any time. When a module is to be loaded, the kernel allocates memory equal to the size of the file, and reads the file into that memory. It then checks the memory for valid modules, adding them to the module directory. Some operating system calls cause a module to be implicitly loaded from disk – for example, if a program fork is requested, and the program module is not already in memory.

Modules explicitly loaded from disk (such as by the **load** utility) are retained in memory even while not being used. For example, commonly used utilities can be loaded, using up memory, but saving on disk accesses. When the module is no longer required it can be removed from the module directory and the memory released, by "unlinking" the module. This is the function of the **unlink** utility. The module header may have the "sticky" (or "ghost") flag set. In this case a module that has been *implicitly* loaded (such as by a "fork" request) will stay in memory after it has been used, until its memory is needed for something else. The Microware-supplied utilities are sticky modules.

Programs must be in modules. A module may be loaded by the operating system anywhere in free memory. Therefore programs (in general) must be position-independent. Their data space may be anywhere in free memory, and is separate from the program space. This allows program modules to be directly ROMmable and re-entrant (as explained earlier).

The operating system itself is broken up into several modules, facilitating customization and expansion. The module system means that no explicit "system build" is required. All modules in ROM and in the boot file are found on startup and installed in the module directory.

3.2 A PROGRAM MODULE

The module header and CRC are automatically generated by the linker when a program (or other module) is assembled or compiled, so there is no added complication to creating programs. The module header consists first of a universal section – the same structure in all modules – followed by a parity check word. After this comes the second part of the header, which varies depending on the purpose of the module, as specified by the module "type" code in the universal part of the header.

As an example, the layout of a program module is shown in figure 5 on the next page. As a module can be loaded anywhere in memory, items within the module cannot be referred to by their absolute memory address. Instead, they are identified by their offset from the beginning of the module header (as if the module were loaded at address 0). In the following description (as throughout this book), a byte is 8 bits, a word is 16 bits, and a long word (or just "long") is 32 bits. A prefix of '\$' indicates a hexadecimal (base 16) number. The offsets from the start of the module header are given in hexadecimal. The C symbolic names are taken from the file 'DEFS/module.h'. There is also an assembly language file 'DEFS/module.a' – the structure item names are similar, though not the same.

3.2.1 Sync Word

The value \$4AFC is an illegal instruction in the 68000 family instruction set, and so occurs very rarely in memory. Starting the module header with this "magic number" speeds up the kernel's search for modules in ROM at coldstart. It checks each word in ROM – only if the word is \$4AFC does the kernel attempt to check the module header parity and Cyclical Redundancy Check. The sync word is also useful to the user when looking through memory.

<u>Offset</u>	<u>Size</u>	<u>C name</u>	<u>HEADER</u>
\$0000	word	_msync	Sync word \$4AFC
\$0002	word	_msysrev	System revision ID
\$0004	long	_msize	Module size
\$0008	long	_mowner	User/group of creator
\$000C	long	_mname	Offset to module name string
\$0010	word	_maccess	Module access permissions
\$0012	word	_mtylan	Type and language
\$0014	word	_mattrev	Module attributes and revision
\$0016	word	_medit	Edition number
\$0018	long	_musage	Offset to usage comments string
\$001C	long	_msymbol	Offset to symbol table
\$0020	word	_mident	Ident code
\$0022		_mspare	12 bytes reserved
\$002E	word	_mparity	Header parity
-----	----		<u>EXTENDED HEADER</u>
\$0030	long	_mexec	Offset to program entry point
\$0034	long	_mexcpt	Offset to default trap entry point
\$0038	long	_mdata	Minimum program data space
\$003C	long	_mstack	Minimum program stack size
\$0040	long	_midata	Offset to data initialization table
\$0044	long	_midref	Offset to data pointers initialization table
-----	----		-----
\$0048			<u>PROGRAM BODY</u>
-----	----		-----
	long		CRC

• **Figure 5 – Module header structure.**

3.2.2 System Revision ID

A number indicating which version of the module header follows. So far the module header structure has not changed, but this field allows for backward compatibility if the header structure is changed in future versions of OS-9.

The current value in this field is \$0001.

3.2.3 Module Size

The size of the whole module, including the header and CRC.

3.2.4 User and Group

This field gives the group number (in the high, or first word of the long word) and user number of the user who created the module. For example, if the user creating the module were user 4 of group 2, this field would contain \$00020004. This allows modules to be protected against use by other users, if desired, by clearing the appropriate flags in the access permissions field.

If a use of a module (link, fork, use in an I/O sub-system) is attempted by the same user (same user ID and group number) as the module creator, the desired modes (read and execute in the **F\$Link** system call, for example) are checked against the private field of the module access permissions. The same check is made if the access is made by a member of the super user group (group zero). If the user has the same group number as the module creator but not the same user ID, the desired modes are checked against the group field of the access permissions. Otherwise, the desired modes are checked against the public ("world") field of the access permissions. If the appropriate permissions flags are not set, the attempted system call fails with a "no permission" (**E_PERMIT**) error.

The kernel also uses the user/group field as a privilege mechanism. Only modules created by a user of group zero (the super-user group) can make certain system calls. The super-user group can compile programs to give controlled access to certain resources by other users, who can run the programs.

3.2.5 Offset to Module Name

The linker adds the desired module name string (terminated with a null character, binary zero) to the body of the module, and sets this field with the offset (from the start of the header) to the name string. The module directory entry contains the address of the module (start of the header). The kernel reads the offset to the name string from the header, adds it to the address of the module, and so can compare the name of the module to the name provided by a program that wishes to locate ("link to") the module.

3.2.6 Access Permissions

Only the low twelve bits of this word are used, as three groups of four bits:

OS-9 MODULES, MEMORY, AND PROCESSES

<u>Bit</u>	<u>Permission</u>
0	Private read (creator only)
1	Private write
2	Private execute
3	reserved
4	Group read (members of creator's group only)
5	Group write
6	Group execute
7	reserved
8	Public read (any user)
9	Public write
10	Public execute
11	reserved

On a system *without* the System Security Module software (and memory management hardware) the flags have the following effect when set:

read	allows load, link, and unlink
execute	allows load, link, unlink, and fork
write	no function

On a system *with* the System Security Module software and memory management hardware, the flags have the following effect when set:

read	allows load, link, and unlink
execute	allows load, link, unlink, and fork
write	allows writing to the module in memory once linked to

A violation of any of these permissions when attempting to gain access to the module (for example, by linking to it) gives an error number 164 (**E_PERMIT**).

3.2.7 Type and Language

The "type" code is in the high (first) byte, while the "language" code is in the low byte. The type code indicates the intended usage of the module. Microware have reserved codes 0 to 15, of which the following codes are currently used:

<u>Code</u>	<u>Module type</u>
1	program
2	subroutine
4	data module
11	trap handler
12	operating system component (other than I/O)
13	file manager
14	device driver
15	device descriptor

The kernel will check the type code against an attempted usage. For example, trying to fork a module that is not type 1 gives an error number 234 (**E_NEMOD**). If a program specifies a type code of zero when trying to link to a module, the kernel will allow the link whatever the actual module type. Otherwise, the type code must match.

Note that it may not be advisable at present for users to make use of the undefined type codes (16 to 255). This is because the "unlink" system call (**F\$UnLink**) assumes that any module with a type code of 13 or higher is part of the I/O system, causing a search of the device table to see if the module is still in use. However, this should not cause a problem, as the module will (presumably) not be found in the device table.

The language code indicates the encoding of the module body. Microware have reserved codes 0 to 15, of which the following codes are currently used:

<u>Code</u>	<u>Language</u>
1	object (machine code) executable
2	compiled Basic intermediate code

Other codes are defined, but have no use at present:

<u>Code</u>	<u>Language</u>
3	compiled Pascal intermediate code
4	compiled C intermediate code
5	compiled Cobol intermediate code
6	compiled Fortran intermediate code

The kernel will check the language code against an attempted usage. For example, trying to fork a module that is not type 1 gives an error number 234 (**E_NEMOD**). If a program specifies a language code of zero when trying to link to a module, the kernel will allow the link whatever the actual module language. Otherwise, the language code must match.

The **shell** uses the language code to automatically fork up an appropriate run-time interpreter for intermediate code programs. For example, if the language code is 2, **shell** will fork the program **runb** to run the Basic compiled intermediate code program.

Note: the kernel will allow multiple modules of the same name in the module directory, provided they have a different type and/or language code. A **link** (or an **unload** - an unlink by name) with a type and language code of zero, will operate on the first module of that name in the module directory.

3.2.8 Attributes and Revision

The module attributes are in the high (first) byte of this field. The revision number is in the low byte.

☐ Module attributes

The attributes are a set of bit flags, indicating special treatment by the kernel:

<u>Bit</u>	<u>Meaning when set</u>
7	module is sharable
6	sticky module
5	supervisor (system) state module

If the module is sharable, the kernel will permit a link count greater than one. This flag is normally set for all modules. If this flag is clear, only one link is permitted at a time. A device descriptor with this bit clear can only

have one path open to it at a time. A program module with this bit clear is not re-entrant – there can only be one incarnation of the program at any one time – which would allow it to be self modifying.

A "sticky" module remains in memory even when its link count has been reduced to zero. A further reduction in its link count (by the **F\$UnLink** system call, for example from the **unlink** utility) causes the module to be removed from the module directory, and its memory returned to the free pool. Also, if the operating system receives a memory request that it cannot satisfy from the existing free pool, it will remove sticky modules whose link count is zero from the module directory (in the order of their entry in the table) until enough memory is available. This mechanism allows commonly used modules to remain in memory after use until they are needed again, or until their memory space is needed.

The "supervisor state" flag is set for all operating system components, and for programs (and trap handlers) that are to run in supervisor (rather than user) state. The 68000 family microprocessors have two operating states – supervisor and user. The processor's internal status register has a bit that indicates supervisor state when set, user state when clear. Certain instructions are not permitted in user state. Any operating system call or external interrupt automatically puts the microprocessor into supervisor state, so all operating system components run in supervisor state. This mechanism provides essential protections between programs in a multi-user system, and (with memory management hardware) prevents programs from corrupting the operating system.

For these protections to work, programs run in user state. In certain applications, however, it may be advantageous to run a program (or trap handler) in supervisor state. This requires great care, and a thorough understanding of the effects (described in the chapter on OS-9 System Calls).

Supervisor state is often called system state, because the operating system runs in this state. In this book the term "supervisor state" refers to the physical operating state of the microprocessor, while "system state" refers to the logical state of the software. For example, a different microprocessor might not have a supervisor state, but the computer would still be logically in system state when executing an operating system function or interrupt service routine.

In OS-9 supervisor state and system state are synonymous. The operating system tests the supervisor state bit of the processor's status register to determine if the computer is in system state.

□ Module revision number

The revision number feature allows modules to be superseded without removing them from memory. It is useful as a means of overriding out-of-date modules in ROM, or in the boot file (the boot file is ROM as far as the operating system is concerned).

When checking a module in ROM on coldstart, or when loading a module into memory, the kernel scans the module directory to see if a module of the same name, language code, and type code is already in the module directory. If so, the kernel compares the revision numbers of the two modules. If the new module has a higher revision number than the module already in the module directory, the kernel overwrites the existing module directory entry with the information about the new module.

If the two modules have the same revision number, the kernel checks that the following conditions are satisfied:

- 1) The system supports "sticky" modules (the **B_Ghost** bit of the **D_Compact** byte of the System Globals is set).
- 2) The link count of the old module is zero.
- 3) The link counts of any other modules in the same module group are zero.
- 4) The new module is not in the same module group as the old module (this check was omitted in OS-9 version 2.2).
- 5) The kernel has finished its coldstart (the **D_ID** field of the System Globals contains \$4AFC).

If all these conditions are satisfied, the kernel frees the memory of the old module and overwrites the existing module directory entry with the information about the new module. If any of the conditions is not satisfied, or the revision number of the new module is lower than the revision number of the old module, the kernel ignores the new module (and returns the memory

used to read in the file, in the case of a **load**), and returns error 231 - "module already known" (**E_KWNMOD**).

It is very important to note that if the new module's revision number is higher than that of the old module, the kernel makes no checks to ensure that the old module is not in use before it deletes the module directory entry. Also, the kernel does not return the memory of the old module to the free pool (because it assumes the module is in ROM). Therefore this feature must be used with care.

3.2.9 Edition Number

This is a software maintenance edition number for the module. It is set by the programmer when the module is created, and is not used by the kernel. It allows a user to identify the edition of a program or operating system component when asking for technical assistance from the software supplier.

3.2.10 Other Fields

☐ Offset to usage comments string

This field is not used at present.

☐ Offset to symbol table

This field is not used at present.

☐ Ident code

The ident code is intended to be used by the **ident** utility when inspecting a module header to display information about the module. **ident** does not use it at present.

3.2.11 Header Parity

This word is the one's complement of a word-by-word exclusive OR of all the preceding words in the header. Therefore the header parity is correct if a word-by-word exclusive OR of all words in the header up to *and including* the header parity gives a result of \$FFFF.

While checking the module header parity prior to installing the module in the module directory, the kernel also calculates a "checksum" over the

module header, and saves this value in the module directory entry for the module. When the kernel receives a request to link to the module, it rechecks this module header checksum, and verifies that the header checksum word now calculated from the header matches the value saved in the module directory. If there is a change, the kernel assumes that the module is corrupt, and returns an error number 236 (**E_BMHP**). This gives some protection against disastrous system errors that might occur if the kernel were to use the entries in the header of a module that has been corrupted after being loaded into memory.

In OS-9 version 2.2 this checksum was simply the low 16 bits of the sum of all the words in the universal module header, including the parity word. In later versions the calculation is slightly more complex - after each word addition the word result is rotated right by a number of bits equal to its own value (modulo 16).

3.2.12 Offset to Program Entry Point

Also known as the "execution offset", this is the offset from the start of the module header to the first instruction to execute in a program. From this the kernel can calculate the absolute address at which to start program execution, by adding this field to the actual address in memory of the module. The first instruction to execute need not be the first instruction in the module - it can be anywhere in the module.

The linker calculates this offset when creating the module, using the entry point offset given in the "root" **psect** in the files used to create the module. The linker can create a program from multiple files, but there must be one and only one file containing a "root" **psect** (described in the chapter on the C Compiler, Assembler, Linker, and Debugger), so no confusion can arise.

The C compiler only produces non-root **psects**. When a C program is linked, the 'cc' executive program adds the file 'LIB/cstart.r' at the front of the list of files to link. 'cstart.r' (created from 'C/SOURCE/cstart.a') is a root **psect**, containing a function to initialize the program and then call the **main()** function of the program.

3.2.13 Offset to Default Trap Entry Point

This offset is produced and used in a similar way to the execution offset. The kernel uses it to calculate the address of the program function to call if the program uses a 68000 **TRAP** instruction for which it has not installed a trap handler module.

3.2.14 Minimum Program Data Space

When creating a module, the linker adds up the static storage (variables) definitions in each of the Relocatable Object Files to calculate the total static storage requirement of the program, and sets this field with that value.

3.2.15 Minimum Program Stack Size

The linker sets this field with an assumed maximum stack usage by the program. As program functions can be recursive (call themselves directly or indirectly), the linker cannot actually calculate a real maximum stack requirement. It uses a default value (3k bytes), which can be overridden by a command line option to the linker.

The kernel adds together the "minimum data space" and the "minimum stack size" fields to calculate the total memory required (initially) by the program. It adds to this the size of the parameter string being passed to the program by the parent process (because the kernel copies the parameter string from the parent's buffer to the top of the child's memory space), and the "additional stack size" parameter passed to the **F\$Fork** system call. The result is the total memory the kernel must allocate for the new process. The low part of the memory allocated is used for the static storage, the middle part for the stack, and the top part for the copy of the parameter string.

3.2.16 Offset to Data Initialization Table

The offset from the start of the module header to a table (built by the linker) of data values for use by the kernel when initializing the program's static storage. The C language permits static storage variables to be initialized with constant values at startup. The kernel uses this table to implement this feature.

The table structure is as follows:

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$0000	long	Offset within static storage to start of area to initialize.
\$0004	long	Size of area to initialize, in bytes.
\$0008	bytes	The initialization data.

The kernel copies the data from the table to the indicated part of the static storage.

The linker separates out all initialized static storage definitions from all uninitialized static storage definitions. It locates the initialized static storage after the uninitialized static storage, so the initialization table is a direct image of the desired initialized variables in memory. Because the linker separates out the static storage definitions in this way, the location of variables in memory may differ from that expected. However, the order of the variables is maintained (but separated into the two areas). This is analogous to splitting a single mixed queue of badgers and foxes into two separate queues.

3.2.17 Offset to Data Pointers Initialization Table

The offset from the start of the module header to two tables (built by the linker) of structures for initializing pointer variables in static storage. The C language permits static storage pointer variables to be initialized with the addresses of other static storage locations, or of program functions.

As the absolute address of neither the program nor the static storage is known when the linker creates the module ("link time"), the linker cannot put the required absolute addresses for these variables into the "data initialization table" described above – it can only put the offset from the start of the module header (for program function pointers) or static storage (for static storage pointers). When the program is forked the kernel (which now knows the address of the program module and of the static storage it has just allocated) must adjust these initialized values in the pointer variables. The kernel locates the pointers that must be adjusted using the information in these two tables.

The first table contains information for initializing pointers to program functions. The kernel must add the address of the start of the module header to the offsets in this table to form the absolute pointer values. The second table contains information for initializing pointers to static storage locations. The kernel must add the address of the start of the static storage to the offsets in this table to form the absolute pointer values.

Each table consists of zero or more lists. Each list has the following format:

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$0000	word	high (most significant) word of the offset to use
\$0002	word	number of entries in the list
\$0004	words	the list – each word is the low (least significant) word of the offset to use

The table ends with a long word of zero – that is, when looking for the next list, the kernel reads the "high word" and "number of entries" values, stopping when the "number of entries" is zero. The second (static storage pointers) table follows immediately after the terminating long word for the first table.

For each entry in the list, the kernel reads the "low word" value and adds it to the common "high word" value (shifted left 16 bits) for the list, to form a long word offset into the static storage, from which it calculates an absolute address within the static storage. It then adds to the long word (pointer variable) pointed to by that address to either the start address of the module header (first table) or the start address of the static storage (second table).

Because the static storage pointer variables are "initialized static storage", they are included in the "data initialization table" described above. This has already initialized them with the offset (calculated by the linker) from either the start of the module or the start of the static storage to the item they are intended to point to. This second operation of adding the start address of the module header or of the static storage changes the offset into the required absolute address.

3.2.18 Module CRC

OS-9 uses a 24 bit Cyclical Redundancy Check value at the end of the module. The linker adds a byte of zero to the end of the module body before the CRC to ensure that the module is an even number of bytes long – the 68000 family of processors requires that all instruction words be on an even address.

The CRC is generated by the linker and checked by the kernel using the "generate CRC" (**F\$CRC**) system call. This call takes an initial accumulator, a byte count, and a pointer to the bytes over which to calculate the CRC. It returns the new accumulator value. Using this call a module's CRC can be generated piece by piece, or all at once. Initially the accumulator must be set to \$FFFFFF. When generating a CRC the final value must be one's complemented before adding the three bytes to the module. The linker generates the CRC over the whole of the module, from the first byte of the header to the zero byte before the CRC. When checking a CRC, the three bytes of CRC must be included in the calculation. The result (for a good CRC) is \$800FE3.

The kernel does not recheck the CRC of a module after it has been installed in the module directory. Therefore the contents of a module (but *not* the

header – see **Header Parity** above) can be altered. This is clearly necessary for data modules (which are pools of data shared between multiple programs). It is also used by the debugger programs for placing "hard" breakpoints in the modules being debugged.

The CRC calculation is not trivial – it requires a significant processor effort (although the time taken to perform the CRC calculation is not noticeable when loading a program, especially with the higher performance members of the 68000 family). Therefore highly time-critical applications should load all the modules they need before starting the application proper. This is good practice in any case, to reduce the possibility of disk usage affecting a time-critical application.

3.3 MODULES IN FILES

Modules may be contained in disk files, just like any other data. One or more modules may be contained in the same file, merged together sequentially in the file. A module – especially a program module – is normally held in a file of the same name as the module, to avoid confusion, but this is not a requirement. To the disk file manager, nothing distinguishes a file containing a module from any other file.

The **load** utility uses the "load" system call (**F\$Load**), which allocates an area of memory equal to the size of the file, reads the file into memory, checks the modules in the file, and installs them in the module directory (using the **F\$ValMod** system call). The module directory entry link count for the first (or only) module in the file is set to one (the others are set to zero).

The **unlink** utility uses the "unlink" system call (**F\$UnLink**), which decrements the link count of a module. If this reduces the link count to zero, the module is removed from the module directory and its memory is returned to the free pool (unless it is a "sticky" module, in which case the module is removed from the module directory if its link count is reduced to -1).

The operating system "fork" call (**F\$Fork**) attempts to find a module of the given name in the module directory. Failing that, it loads a file of the same name from the execution directory, and forks the first module in the file (*which could have a different name*). Note: if a module of the right name exists in the module directory, but it is not type "program" or language "object code", the kernel rejects the "fork" request with the error number 234 (**E_NEMOD**), rather than trying to load a file of the given name.

Exiting from a program causes the program module to be unlinked by the kernel. Therefore if it was implicitly loaded by a "fork" its link count is reduced to zero, and it is removed from the module directory (unless it is a sticky module).

3.3.1 Module Groups

OS-9 allocates memory in multiples of a minimum allocation unit (at present 16 bytes). This is because the areas of free memory are connected together in a linked list, with each free memory area having a controlling structure at the start of the memory area. This structure needs a certain amount of memory, so an area of memory smaller than this could not be freed (de-allocated) – it could not be returned to the free memory list.

However, a file containing more than one module could contain modules whose length is not an integral multiple of this minimum block size, so that when the modules are loaded together in memory there may be memory blocks that contain the end of one module and the start of the next. Therefore, if the memory used by each such module could be freed separately, the free memory areas would no longer be integral multiples of the minimum block size, which cannot be permitted. To solve this problem, Microware devised the concept of the "module group".

Multiple modules loaded from one file constitute a "module group". The address of the first module (and therefore of the memory area allocated to the group) is used as a unique "group identifier" for the group. Each module in the group has this identifier in its module directory entry. All modules in the group remain in the module directory (even if they individually have a link count of zero) until the link count of all of them goes to zero.

That is, when – as a result of an "unlink" – the link count of a module goes to zero (or is already zero, and so would go to -1, for a sticky module), the kernel checks all entries in the module directory to see if there are any other members of this module group whose link count is not zero. If so, it does not remove the module from the module directory. Otherwise, it removes all modules in the group from the module directory, and frees the memory for the whole group.

Note that the kernel does not check whether the other modules in the group are sticky modules. Also, if a module (including a sticky module) is unlinked when its link count is already zero, but another module in the group has a non-zero link count, the link count of the first module remains zero.

When a file of modules is loaded, the kernel sets the link count of the first module to one, and the others to zero. Therefore if no use is made of the modules (so their link counts are not increased), unlinking the first module will remove all modules in the group from the module directory.

Modules in ROM and the boot file form module groups of their own (a group for each separate ROM area in the memory lists), with a group identifier equal to the address of the ROM area containing the modules. During its coldstart the kernel scans the ROM (including the boot file in RAM, which the boot program pretends is ROM to the kernel) looking for modules, and puts them in the module directory. It sets the link count of the first module in each group to one. The kernel then links to itself, so setting its link count to one. This holds each group in the module directory. If each module in a group were unlinked so its link count reached zero, all the modules in the group would be removed from the module directory (as for any group), with potentially fatal consequences.

There is very little protection on module unlinking. Unlinking a module that is in use can have fatal consequences. The kernel does not permit the link count of an I/O module (file manager, device driver, or device descriptor) to be reduced to zero if the module is in use by any entry in the device table. Also, OS-9 version 2.4.3 (released in 1992) does not permit the link count of a module to be reduced to zero if it is the primary program module of any existing process, or an installed trap handler module of any existing process.

3.4 OS-9 MEMORY

The kernel provides dynamic allocation of memory. That is, memory is allocated as needed, and when it is no longer needed it is returned to a free pool.

OS-9 does not use memory management hardware to translate physical memory addresses to logical addresses as seen by a program. The program "sees" the memory at its actual physical address. Therefore a request for memory cannot know where the memory will be located. This means that all system memory usage must be register indirect – that is, relative to a processor internal register that has been loaded with the base address of the allocated memory area.

The same simple mechanism is used for allocating operating system memory and program memory, irrespective of whether the request is made from a program or from an operating system component.

A program is initially allocated memory for its static storage variables and stack by the kernel when the program is forked. In addition, the program may dynamically allocate and release additional areas of memory of any size (rounded up to the minimum allocatable block size), up to a maximum of 32 areas at any one time (including the process's static storage). This limit is imposed because the kernel must keep track of the memory areas owned by the program, so that the program memory can be automatically returned to the free pool on program exit (in case the program is abnormally terminated, or does not clean up before exiting).

If a newly allocated memory area is contiguous with an area already allocated to the process, the two areas are merged – only one entry in the table is used. If the newly allocated memory area fills a hole between two areas already allocated to the process, the three areas are merged into one table entry. This approach makes the maximum use of the 32 entries available in the table.

Because a memory management unit is not used to translate addresses, the kernel cannot combine separate physical memory areas to make a single logical memory area. Therefore problems due to memory fragmentation can occur, but in practice they are very rare, unless the application is very tight on memory, or is allocating very large blocks of memory.

The **mfree** utility used with the '-e' option displays the list of currently free memory areas.

3.4.1 Coloured Memory

Note: "coloured memory" and the "memory list" in the **init** configuration module were added to OS-9 in OS-9 version 2.3. Earlier versions of the operating system do not have these features.

The coloured memory concept was devised to solve two problems:

- a) It is desirable to be able to control the order of allocation of different memory areas, as some memory areas may be slower in operation than others – that is, memory areas need to have a **priority** number.

- b) Some areas of memory may have special properties (for example, a memory area that is accessible to another processor on the same bus), so it is desirable to be able to specifically reference these different types of memory when allocating memory – that is, memory areas need to have a **type** number.

Microware call the different memory types "colours" – hence the term "coloured memory". Giving a special type of memory a unique colour number allows programs to allocate memory of that type without needing to know the absolute address of the memory. It also allows multiple programs to allocate separate areas of the special memory. Furthermore, if the system has memory management hardware which OS-9 is using for inter-task memory protection, a program cannot access memory which has not been allocated to it – coloured memory is a convenient way for a program to gain access to special memory in a way that is guaranteed to be portable to other OS-9 systems. Examples of memory with special properties are:

- Graphics display memory.
- Battery-backed memory.
- Inter-processor communication memory.

The need for prioritizing the allocation of memory areas is quite common in bus-based computers. The memory on the processor board is usually much faster (for the processor to access) than memory on a separate memory board accessed over the bus.

The kernel builds its table of memory areas from the list of possible memory areas in the **init** configuration data module. During coldstart the kernel reads this list, and tests each area in the list, to see how much memory actually exists in that area. Each entry in the list specifies a start address for an area, an end address, a colour number, a priority, and attribute flags such as "read only", for ROM areas that are to be checked for modules on startup, and "user" for memory that can be allocated to user-state programs.

This allows the user to specify the colour, priority, and attributes of each memory area that may be in the system, and to "hide" memory from OS-9 (by not including it in the list). There are two memory allocation system calls – general (or "uncoloured") and coloured. The uncoloured memory allocation system call is **F\$SRqMem**, called by the C library function **_srqmem()**. The coloured memory allocation system call is **F\$SRqCMem**, called by the C library function **_srqcmem()**. In addition, the system calls to load modules from a file (**F\$Load**, C functions **modload()** and **modcload()**), and to create

a module in memory (F\$DatMod, C functions `_mkdata_module()` and `make_module()`), provide an extended format that specifies the memory colour to use (see the chapter on the OS-9 System Calls).

If there is more than one memory area of the same colour, memory with a high priority will be allocated before memory with a low priority. The uncoloured memory allocation system call allocates memory in priority order irrespective of colour, and will not allocate memory with a priority of zero. Such memory can only be allocated by a coloured memory allocation request, and so is protected from general system usage.

3.4.2 Memory Allocation

OS-9 uses a simple but effective algorithm for the allocation and de-allocation (freeing) of memory. The kernel maintains a separate linked list of free memory areas for each priority value of each colour of memory. Each free memory area has at its start the following structure:

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$0000	long	Address of next area in linked list.
\$0004	long	Address of previous area in linked list.
\$0008	long	Size of this area (in bytes).

To keep track of these linked lists, the kernel maintains a table – the memory colour node table – in which each entry is a structure, describing each memory area found at startup. The structure gives the memory area start and end addresses, the memory area colour number, allocation priority and attributes, and the total size of the free memory areas within this memory area. It also gives the addresses of the first and last free memory areas in this memory area.

The memory colour node structures in the table are linked together as a doubly linked list, ordered by allocation priority (highest priority area is first in the list).

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$000	long	Start address of memory area.
\$004	long	End address plus one of memory area.
\$008	long	Address of next (lower priority) memory colour node in list.
\$00C	long	Address of previous (higher priority) memory colour node in list.

OS-9 MODULES, MEMORY, AND PROCESSES

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$010	long	Address of first free memory area within this memory area.
\$014	long	Address of last free memory area within this memory area.
\$018	long	Reserved.
\$01C	long	Start address of memory area as seen by alternate bus master (local start address plus translation offset given in memory list in <code>init</code> module).
\$020	long	Sum of sizes of free memory areas in this memory area.
\$024	word	Attributes of this memory area (as given in the memory list in the <code>init</code> module).
\$026	word	Colour number.
\$028	word	Priority.

So, to allocate memory of a given colour the kernel walks through the linked list of colour node structures until it finds one of the correct colour that has sufficient free space. From the structure it takes the addresses of the first and last free memory areas within this memory area. It then walks through that linked list of free memory areas, checking the size of each entry until it finds an area big enough to satisfy the request (rounded up to the nearest minimum allocatable block size). If the area is bigger than the request, the kernel allocates the amount requested from the top of the area. The kernel reduces the size value in the area's information structure by the requested amount, but does not need to alter the linked list.

If there is no single free memory area in that linked list large enough to satisfy the request, the kernel returns to the table of colour node structures, and continues walking through it to find another area of the desired colour. This allocates memory of the desired colour, using high priority memory of that colour first.

An "uncoloured" memory allocation request is a request for memory of any colour, provided the memory priority code for the area is not zero. In this case the kernel walks the list of colour nodes without regard for the colour number in the descriptor, and so allocates the memory by priority only. The search stops if the colour node has a priority value of zero – such memory areas must not be allocated by an uncoloured memory request, and all further descriptors in the list will also have a priority of zero.

If the area is exactly the right size for the request, the kernel simply removes the area from the linked list.

Note: memory for the primary data space of a process, allocated by the kernel by the **F\$Mem** system call when forking the program, is a special case. It is allocated from the *bottom* of a memory area.

3.4.3 Inter-task Memory Protection

Memory management hardware (usually) performs two functions. It translates logical memory addresses generated by the processor into different physical addresses to the memory chips, and it also provides protection against illegal memory accesses by generating a "bus error" signal to the processor if a particular memory access is not permitted. An access may be forbidden either because the operating system has set the "memory map" for the currently executing program in the memory management hardware to exclude that area of memory, or because the program has attempted an operation which the memory map specifies is not permitted (such as writing to an area that has been marked as "read only" in the memory map).

By changing the memory map in the memory management hardware for each program that runs, the operating system can prevent programs accessing memory that has not been specifically allocated for their use. This prevents system corruptions or crashes due to incorrect memory accesses resulting from programming errors.

OS-9 does not use the address translation capability of memory management hardware, because OS-9 is capable of running without the need for such hardware. But OS-9 can use memory management hardware to provide inter-task (that is, "between programs") memory protection. This feature of OS-9 has been available as an option for use with any memory management hardware from OS-9 version 2.2 onwards, and is standard with OS-9/68030 and OS-9/68040 (because these processors have a built-in memory management unit).

To avoid dependence on a particular memory management unit (MMU) chip, the kernel does not contain any functions to handle the MMU. Instead, these functions are provided in the System Security Module **ssm**. If **ssm** is not in ROM or the boot file (so it is not present during the kernel's coldstart), or is not in the list of "kernel customization modules" in the **init** configuration module, the kernel assumes that there is no MMU, and does not implement inter-task memory protection. Therefore to disable inter-task memory protection it is only necessary to leave **ssm** out of the boot file (and ensure it is not in ROM).

When a process is first forked, its memory map contains only its program module and its primary data space (static storage and stack). If the process allocates additional memory, that memory is added to the memory map of the process. (The program is the body of instructions in the program module. The process is this "incarnation" of the program, with its own data space and other resources). Similarly, if the process subsequently de-allocates the memory, it is removed from the process's memory map.

OS-9 can allocate memory logically in multiples of its "process minimum allocatable block size" - at present, 16 bytes. However, most memory management units cannot manage memory in such small blocks. The memory maps within an MMU will be built in larger blocks - 4k bytes is a typical size. This block size is known to OS-9 as the "system minimum allocatable block size". Physically the kernel must allocate memory to a process in multiples of this block size. However, this could waste a lot of memory if the program were to make a number of small memory requests, and the kernel allocated a separate block for each request.

Therefore, if a program makes a small memory request the kernel allocates (as it must) a whole block. But it adds the remainder of the block to the linked list of free memory fragments belonging to the process. If the program makes another small memory request, the kernel will try to use memory from the free fragments of the already allocated blocks. Only if the program makes a memory request that cannot be satisfied from the allocated blocks does the kernel allocate a new block (or blocks).

The kernel uses the same mechanism for managing memory requests from operating system components - it keeps track of the free fragments in the blocks allocated to the operating system. This is particularly important in saving memory, because the kernel itself allocates many relatively small memory areas for use as tables and resource management structures.

Note that if inter-task memory protection is not being used (SSM is not present), the kernel still uses the same technique of allocating fragments of memory from a larger block. In this case the system minimum allocatable block size is set to 256 bytes.

When a process links to a module (or creates a data module in memory), the memory area of the module is added to the process's memory map, so the program can access the module (including the module header). The module attributes (write permit in the owner, group, and public fields) determine whether the process can write over the module. Data modules are therefore a

very useful mechanism for multiple programs to share a common area of memory.

If inter-task protection is used, then user state processes cannot access the registers or memory of input/output interface chips and circuits directly. This may be seen as a restriction when building a simple application that must perform some hardware control. Therefore the **ssm** adds system calls to the operating system that permit a program (provided it is a member of group zero) to add specific areas of memory to its memory map. These system calls (**F\$Permit** and **F\$Protect**) are described in the chapter on the OS-9 System Calls.

The operating system (and system state processes and trap handlers) have unrestricted access to all memory. The **ssm** configures the MMU to suspend memory protections for processor accesses in supervisor state.

3.5 PROCESSES AND MULTI-TASKING

A process consists of a program that has been forked and has not exited (or been abnormally terminated), together with its data memory and a controlling memory structure used by the operating system to manage the process. This structure is known as a "process descriptor". Because OS-9 strongly suggests that programs be re-entrant, a single program may have any number of "incarnations" at any one time, each using the same program module, but having separate data memory. Each such "incarnation" is a separate process. The kernel maintains a separate process descriptor for each process, which it uses to control the process, and to retain information about the process. The process descriptor is described in the chapter on the OS-9 Internal Structure.

A process (or task) is created by a "fork" request to the operating system. Note that the word "task" is effectively synonymous with "process", and the two words are often used inter-changeably. The "fork" system call (**F\$Fork**) returns a number, known as the process ID, that uniquely identifies the new process. The process ID is used for any system calls that communicate with or modify the behaviour of the process. Note that once a process has died its process ID may be assigned to a subsequently forked process, but no two processes will have the same ID at the same time. The process ID is always greater than 1 - zero is not used, and the System Process has the process ID of 1.

Under OS-9, each process has a "process priority" value associated with it. This value is assigned to the process when it is created ("forked"), and is

usually the same as the priority of the process that forked it (the "parent"). The priority of a process can be changed by the **F\$SPrior** system call, and is the main mechanism for determining what share of the processor's time a process will get. The use of the process priority value, and the operation of the OS-9 process scheduler, are described in the chapter on Multi-tasking.

In a typical real time application, many processes work together to carry out the application. To do this they must exchange data, messages, and synchronization information. These functions - which are fully supported by OS-9 - are known as "inter-process communication". The OS-9 inter-process communication facilities are described in the chapter on Inter-process Communication.

At any given time, a process will be in one of the following states:

Active	Requesting processor time.
Waiting	Waiting for a child process to die.
Sleeping	Waiting for a timed period, or an external (hardware) event, or an inter-process communication signal.
Waiting for event	Waiting for an inter-process communication event.
Debugged	Waiting for its parent (a debugger program) to permit it to continue execution.
Dead	waiting to report its exit status to its parent.

Processor time is divided up between the currently active processes in time units known as "time slices". This is achieved by means of a hardware timer that produces processor interrupts at regular intervals known as "ticks". A time slice is one or more ticks. Most OS-9 systems use a 10ms tick, with two ticks per time slice. Two ticks per time slice are used rather than one, because the operating system cannot resolve time in units less than one tick. The kernel will assume that a part tick is a whole tick, so in effect a time slice of two ticks is actually between one and two ticks. A time slice of one tick could in reality be vanishingly small!

It is the execution of each process in turn for a short period of time that gives the appearance of programs executing concurrently. Therefore the length of a time slice is chosen to be short enough to give an acceptable appearance of

concurrency (for the application), yet not so short that the operating system is spending too much time in scheduling the processes. Only active processes receive processor time, so no processor time is wasted. The process scheduler of OS-9 (described in the chapter on Multi-tasking) ensures that the processor time is divided equally and evenly between all the active processes, unless the user or programmer requests otherwise – OS-9 offers several different mechanisms for the user or programmer to modify the behaviour of the process scheduler. In addition, the OS-9 scheduler has certain special features to ensure the response of high priority processes in real time applications, and to improve the throughput of the I/O system. These features are fully described in the chapter on Multi-tasking.

The **procs** utility shows all processes currently existing on the system, and displays additional information about their past performance and current state.

A process is started by a "fork" system call (**F\$Fork**) from another process or an operating system component. It finishes when its program "exits" (**F\$Exit** system call), or when the process encounters a fatal condition (a signal or processor exception it cannot handle). The use of signals is covered in the chapter on Inter-process Communication, while processor exceptions are described in the chapter on Exception Handling. The parent process can, in its "fork" request, pass a pointer to a memory area – known as a "parameter string" – which is copied to the new process's static storage. The parent can also specify the process priority of its new child, and request that it be allocated more static storage than the minimum specified in the program's module header.

A process can also transform itself into a new process, executing a different program. This is done by a "chain" system call (**F\$Chain**). This system call is very similar to a fork, but the calling process is terminated and its process descriptor is used for the new process, which therefore has the same process ID as the original process.

Any number of processes may exist at any one time. The kernel keeps track of them by using the process descriptors. It keeps track of the process descriptors by means of the "process descriptor table", which is an array of addresses of process descriptors. The process ID of a process is an index into this table. If the entry in the table is zero, that process ID is not currently in use for any process. The process descriptor table starts off small, in order not to waste memory, but is extended by the kernel if it becomes full – the kernel allocates a new table of twice the size, and copies the old table into the new table.

3.5.1 A Dead Process

Every process initially has a parent – the process that forked it. A process may be "disinherited" (lose its parent), usually because the parent dies before the child. It would be possible for a process to be disinherited without the parent dying, by reorganizing the relevant links between the process descriptors, but there is at present no system call to do this.

A process that dies and has not been disinherited returns its exit status to its parent. This can only occur when the parent makes a "wait" system call (**F\$Wait**), to "wait for child to die". Therefore a "dead" process can remain hanging around indefinitely until its parent dies or executes a "wait" request. The kernel de-allocates all of the resources of a dead process (memory, I/O paths, program module), but retains the process descriptor until the parent dies or makes a "wait" request to receive the child's exit status.

This guarantees proper operation in a multi-tasking application, where it may be essential for the parent to know the exit status of the child. However, it can cause some confusion to a user, because the user expects that when he has killed a process, it has gone.

The **shell** provides two commands to manage the death of a process. The **shell** 'w' command will wait for any one child of the **shell** to die before returning to the prompt. The **shell** 'wait' command will wait for all children of that **shell** to die.

There is no system call to disinherit a child, allowing the parent to just forget about it. One way to achieve the same effect in a multi-tasking application is to fork up a process that then forks up the desired child and dies. This leaves the child disinherited (it does not become a child of the "grandparent").

3.5.2 System State Processes

Programs normally execute in 680x0 user state. The operating system components always execute in system state (the supervisor state of the processor). In user state, certain operations such as the masking of hardware interrupts are considered illegal by the processor. Also, a process may be scheduled out (stop executing, to allow another process to execute) at any point in the program, whereas scheduling is deferred while the processor is in supervisor state (permitting system calls to be indivisible).

Programs may wish to directly handle interrupts, or to prevent themselves being scheduled out during critical code fragments. It is a relatively simple matter to add a system call (in a device driver, or a kernel customization

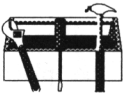
module) to mask and unmask interrupts, and facilities exist in the operating system to modify or pre-empt the normal multi-tasking scheduling mechanism.

Another method, however, is to run the process in system state. This can be achieved by setting the "supervisor state" flag in the program module attributes byte in the module header. When the program is forked, the operating system executes the process in system state, and it has all the privileges of the operating system, such as deferred scheduling.

However, it also has all the responsibilities of the operating system. A thorough understanding of the operation of the operating system is recommended before writing system state programs. For example, many of the system calls (especially the I/O calls) operate somewhat differently if the call is made from system state than if it is made in user state.

CHAPTER 4

THE OS-9 UTILITIES



This section is an overview of the main utilities available with Professional OS-9, to help the user quickly identify which utility he will use for a particular job. It does not attempt to describe the operation or syntax of the individual utilities, nor are the lists of utilities exhaustive, as this information is available in the OS-9 User's Manual.

4.1 WHAT IS A UTILITY?

A utility is a program that performs some function to facilitate the use of the computer system, such as directory display, file copying, or disk formatting. By contrast, an application program performs some function that is an end in itself, such as word processing, data logging, or machine control. Utilities can be divided into three functional groups (although the distinction can be somewhat blurred):

- a) Access to operating system functions from the terminal – for example, **load** to load a file of modules.
- b) System management – for example, **tsmon** to allow the login of a user at a terminal.
- c) General-purpose functions.

Because utilities are just programs, the user can write new utilities, or purchase additional utilities, either from Microware or from third-party suppliers.

4.2 UTILITY SYNTAX

OS-9 utilities have a common command line syntax that is simple to learn and use. The command line starts with the name of the utility. The remainder of the line consists of items separated by spaces or commas, and can contain only two item types (there may be any number of each type, in any order):

- a) An option string, preceded by a '-' character.
- b) A parameter, not preceded by a '-' character.

For example:

```
$ dir -ur fred
```

The '-ur' is a string of two options, 'u' and 'r'. The 'fred' is a parameter.

The utility scans the command line twice, first looking for options, and then looking for parameters. Therefore, in general options may be placed anywhere in the command line in any order. However, parameters may need to be in a defined order. For example:

```
$ copy fred henry -r
```

copies the file 'fred' to the file 'henry', overwriting any existing 'henry'. The **copy** utility determines which is the source file and which is the destination from the order of the parameters on the command line.

An option is used to change the default behaviour of the utility. The option is usually a single character, although some options are two characters. Usually the letter case of the option is not significant, but some utilities have so many options that some options have a different meaning in upper and lower case.

Options may themselves take a parameter. In this case the parameter directly follows the option character, with an optional '=' separating them. Multiple options characters may be grouped after one '-', but an option that takes a parameter must be at the end of the group. This applies even if the option parameter is optional. For example, the **pr** utility accepts the option '-z', to indicate that it should read file names from its standard input path. This option takes an optional parameter - the name of a file to read (instead of reading standard input). Therefore the '-z' option must always be the last in a string of options, even if the first form (no file name) is used.

```
$ pr -tz
$ pr -tz=file_list
```

In the example below, **pr** would assume that 't' is the name of the file containing the list of file names, not the '-t' option character:

```
$ pr -zt
```

Note that if a parameter or option string is to contain a space, comma, or one of the **shell** special characters:

```
( ) # ^ & ! < > * ?
```

it must be enclosed in single or double quote marks. The whole string must be enclosed. For example:

```
$ format -r /d0 -i=3 "-v=My Disk" -nv
```

The options that cannot take a parameter can be merged into one string:

```
$ format -i=3 /d0 "-nvr=My Disk"
$ format -nvri=3 /d0 '-v=My Disk'
```

All well-behaved utilities will respond to the '-?' option, causing them to display helpful usage text rather than carrying out their normal function.

Utilities are written to make as few assumptions as possible about the objects they work on, to increase their usefulness. For example, most utilities can take their input from - and send their output to - any file, device, or pipe, by using the redirection capabilities of the **shell**. Pipes allow the functions of utilities to be combined. For example:

```
$ dir FRED -u ! grep -v "\.r$" ! pr -z >/p
```

The **dir** utility will list all files in directory 'FRED'. The **grep** utility filters that list, removing all file names ending in '.r'. The **pr** utility reads in the file names remaining, and prints each file to the device '/p'.

4.2.1 Formal Syntax Notation

Microware use a standard notation when describing the syntax for the options and parameters of a utility.

[]	the enclosed item(s) is optional
{ }	the enclosed item(s) is optional, and any number may be entered
< >	the enclosed word or phrase is a description of the item to be entered
	separates possible choices

The item descriptions enclosed between '< >' characters are usually abbreviations. Example abbreviations (with examples of usage) are:

<u>Abbr.</u>	<u>Meaning</u>	<u>Example</u>
opt	option string	-z=file_list
path	pathlist	PROJECT/SOURCE/test.c
dir	directory name	PROJECT/SOURCE
name	symbolic name	apricot
str	text string	hello world
n	decimal number	1234

For example, the syntax for the **copy** utility might be described as:

```
copy {<opt>} <source path> [<source path>] | [<destination path>] {<opt>}
```

This indicates that zero or more options are permitted, placed anywhere on the command line, that there must be one source pathlist given, but any number more are permitted, or that optionally a destination pathlist may be given after the source pathlist. (If there is more than one source pathlist, a destination pathlist cannot be specified - the '-w' option must be used to specify a destination directory.)

4.3 UTILITIES FOR OPERATING SYSTEM FUNCTIONS

Many utilities exist to provide access to operating system functions. These are generally simple programs, converting the options and parameters given on the command line into appropriate operating system calls.

attr	Display or change file attributes.
copy	Copy files (but not directory structures).
date	Display the date and time.
deiniz	Detach an I/O subsystem (see the chapter on the OS-9 I/O System).
del	Delete file(s).
deldir	Delete complete directory structures.
dsave	Copy a complete directory structure.
dump	Display file contents in hexadecimal and text.

echo	Print a string to standard output (including binary data).
fixmod	Check or update module CRC and module header items.
free	Display disk free space.
ident	Display information about module(s) in a file or in memory.
iniz	Attach an I/O subsystem (see the chapter on the OS-9 I/O System).
link	Increment the link count of a module in memory.
list	Display a text file.
load	Load modules from a file into memory.
mkdir	Make a new directory.
mdir	Display the module directory.
merge	Concatenate files to standard output.
mfree	Display the free memory map.
pd	Print the current execution or data directory pathlist.
pr	Output files with pagination (for printing).
printenv	Print the environment variables.
procs	Display existing processes.
rename	Change a file name.
save	Save modules from memory to a disk file.
shell	The command line interpreter.
setime	Set the date and time.
sleep	Sleep for a time (in ticks or seconds), or indefinitely.
tee	Copy standard input to standard output and multiple output paths.
tmode	Change options on standard input, standard output, or standard error path (terminal or printer).

THE OS-9 UTILITIES

touch	Set the "last modified data and time" of a file to "now".
unlink	Decrement the link count of a module.

4.4 SYSTEM MANAGEMENT UTILITIES

There are several utilities to assist with maintaining the system, including utilities to archive data, check operating system structures, and manage a multi-user system.

backup	Copy the whole of a disk.
dcheck	Check disk filing structure integrity.
devs	Display a list of currently initialized devices.
format	Format a disk (physical or just logical format).
frestore	Retrieve files from a tape (or disk) archive.
fsave	Archive to a tape (or disk).
irqs	Display a list of currently installed interrupt handlers.
login	Log in a user.
os9gen	Install a boot file on a disk (make the disk bootable).
tsmon	Monitor a terminal for user request to log in.
xmode	Examine or modify the options in a terminal or printer device descriptor in memory - affects all <i>subsequently</i> opened paths on the device.

4.5 GENERAL UTILITIES

Finally, there are many utilities that are simply useful general-purpose programs.

binex	Convert binary data to Motorola S-record format.
build	Copy input lines to a file.

cfp	Command line processor – repeat a command, substituting strings from standard input.
cmp	Binary comparison of two files.
code	Print the hex value of input characters.
compress	Compress a text file.
count	Count the lines in a file, or display a breakdown of the characters.
edt	Simple line editor.
exbin	Convert Motorola S-records to binary data.
expand	Decompress a text file (see compress).
grep	Filter input lines, passing lines that match (or do not match) a given pattern.
help	Display usage information for a utility (calls the utility with the '-?' option).
make	Compile/assemble/link multi-file programs.
qsort	Quick sort of lines in a text file by fields.
tr	Transliterate characters.
umacs	Screen editor – an implementation under OS-9 of the public domain emacs screen editor.

CHAPTER 5

SYSTEM MANAGEMENT

5.1 THE SYSTEM MANAGER



Every computer system – even a single-user system – should have a user designated as the System Manager. The System Manager has responsibility for:

- a) Allocating user IDs, groups, and passwords.
- b) Creating and amending the system startup procedure.
- c) Formatting and maintaining the system disk (the main disk drive).
- d) Maintaining backups and archives of the system disk.

The System Manager is always the super-super user – user zero of group zero. Many system functions are only available to the super user group, or even only to the super-super user. Modules created by the super-super user may only be loaded from files owned by the super-super user. This gives protection against super-user facilities being used illegally.

The initial user ID and group is 0.0. Therefore on a single user system the user is always the super-super user, and has access to all facilities. When a user is logged on, the user ID and group are taken from the password file, according to the user's name and password. So on a multi-user system users can be prevented from unauthorized access to resources, or modification of other users' filing systems.

5.2 THE FILING SYSTEM

Disk files have read, write, and execute permissions for private and public use. For historical reasons the disk file manager ('RBF') does not support separate group permissions, unlike the other components of the operating system. The private permissions relate to use by the file owner *and* other members of the same group. The public permissions relate to all other users. Therefore a file that has only private read and write permissions can only be read, modified, or deleted by members of the file owner's group. Note that the same historical reasons – compatibility with the OS-9/6809 filing system – dictate that the disk filing system can only store the user ID and group as byte values, whereas elsewhere they are stored as word values. This imposes a practical limit of 255 on user IDs and group numbers.

To modify the attributes (permissions) of a file – for example, using the **attr** utility – a user must have write permission for the file. The System Manager has the special ability to modify the attributes of any file. Therefore by doing so he can gain access to any file, and read, modify, or delete it. Note that directories are files, and have file attributes. If a user does not have permission to access a directory, he cannot access any of the files in the directory.

The **dcheck** utility checks the integrity of the filing structure on a disk, and can make simple repairs. Note that sectors found to be bad during the verify pass of disk formatting will be reported as "allocated but not in the file structure".

5.3 THE PASSWORD FILE

The password file is the file 'SYS/password' in the root directory of the "initial mass storage device" specified in the **init** module (usually '/dd' or '/h0'). It should only have permissions for private read and write – for the System Manager (group zero) – to prevent modification by other users. The password file is a simple text file, that can be created and modified by any text editor, such as the **umacs** screen editor. There is a single line entry for each user. The elements on the line are separated by commas. For example:

```
Henry,penguin,1.3,100,.,./dd/HENRY,shell -p="@Henry: "
```

```
Henry                                User name.
```

```
penguin                             User password.
```

1.3	User group number and user ID (separated by a full stop '.') - user 3 of group 1.
100	Initial process priority. Initial execution directory - no change from current.
/dd/HENRY	Initial data directory.
shell -p="@Henry: "	Initial command line to execute - fork the shell command line interpreter, with the '-p' option to set the prompt string.

The initial directories are relative to the directories of the **login** utility processing the password file, which in turn will have inherited the directories of the **tsmon** utility that normally monitors a terminal and forks **login** when the [CR] key is pressed.

When assigning process priorities, bear in mind that a small absolute difference in priorities has a large effect on the allocation of CPU time.

5.4 SYSTEM STARTUP

After going through its coldstart procedure, the kernel forks up the program whose name is given in the **init** configuration data module - usually **sysgo**. The **sysgo** supplied by Microware forks a **shell** to execute a text file 'startup', and then goes into an endless loop forking up a **shell** and waiting for it to die.

The source of **sysgo** (in assembly language) is supplied in the file 'SYSMODS/sysgo.a', to allow customization for special applications.

The 'startup' file - located in the root directory of the initial mass storage device specified in the **init** module - is the place to put commands to load additional modules not present in the boot file, request the date and time (if the system does not have a battery-backed clock chip), and fork any required incarnations of **tsmon**, for multi-user systems. For example:

```
-nt
* The next line is needed if the system does not have a
* battery-backed clock chip:
setime </term
* Load a 'dd' alias device descriptor for the hard disk:
```

SYSTEM MANAGEMENT

```
load BOOTOBS/dd.h0
chx /dd/CMDS
chd /dd
mfree
echo Starting up /t1, /t2, and /t3
echo "Hit RETURN to log on" ! tee /t1 /t2 /t3
tsmon /t1 /t2 /t3 &
* The next line asks for a logon on the system console.
* In this case the 'startup' shell never terminates:
tsmon /term
```

5.5 THE .LOGIN FILE

When **login** (forked by **tsmon** when [ENTER] is hit) forks the initial command line shown in the password file, it does so with a special parameter that causes the shell to look for a file '.login'. (Note that all files whose names start with '.' are normally hidden from **dir**, but may be viewed with the '-a' option). If the file is present in the current directory the **shell** executes the command lines in the file before giving the user the prompt.

This is a very important mechanism, as it allows environment variables to be set, and the default directories to be changed. For example:

```
setenv TERM vt100
setenv _sh 0
setenv HOME ../PROJECTS/PENGUINS
chd
echo "You are in: " -r
pd
```

The environment variable **TERM** is used by screen-orientated programs, such as **umacs**, to determine the type of terminal that is being used. The environment variable '_sh' sets the initial "shell level", used by the **shell** when the first character of the prompt string is '@'. When a **shell** is forked, it looks for this environment variable. If it exists, the **shell** increments its value, and substitutes it for the '@' character in the prompt string (unless it is zero, in which case the '@' is simply not displayed). This helps keep track of **shells** forked from other programs, such as **umacs** or **debug** – the prompt string appears with a leading number if the **shell** has been forked from within another program.

5.6 DISK FORMATTING

Disk formatting is achieved using the **format** utility. **format** has three phases:

- 1) Physical format.
- 2) Verify.
- 3) Logical format.

The physical format phase issues a format request to the device driver, to physically rewrite the sectoring information on the disk. The verify phase reads all the sectors on the disk, to determine which are faulty. The logical format builds the disk identification sector, the allocation bitmap, and the root directory. It effectively "forgets" all files previously existing on the disk.

The physical format and verify phases may be omitted. However, it is recommended that the verify pass always be performed, unless you are certain the disk has no errors, or the drive has automatic defect handling (all modern SCSI hard disk drives have).

The **format** utility '-c' option allows the specification of a "cluster size" other than the default of one (it must be a power of two). The cluster size is the number of sectors per bit in the allocation bit map, and is the minimum allocatable block of disk space. RBF uses the bulk of the File Descriptor sector of a file for the file's segmentation table. Each entry takes 5 bytes (a 24-bit start logical sector number, and a 16-bit number of sectors). For example, if the sector size is 256 bytes, the table can accommodate 48 entries, each referring to a maximum of 65535 sectors.

However, there is a further restriction on allocation of space in a file. RBF will not allocate a segment for which the allocation would cross a bit map sector boundary. This limits a segment to a maximum of "eight times the sector size" clusters. For example, if the sector size is 256 bytes, a segment cannot exceed 2048 clusters. At this sector size a file cannot have more than 98304 clusters (approximately 24Mbytes at one sector per cluster). Therefore it is recommended that larger disks be formatted with a cluster size greater than $1 - \text{approximately "disk size in Megabytes" divided by } 10$, if the sector size is 256 bytes.

5.7 INSTALLING A BOOT FILE

On systems where the operating system is not in ROM, the boot program reads a file known as the boot file from disk. This file contains at least the basic operating system. To simplify the boot program, special information is put on the disk to identify the boot file. This is done using the utility **os9gen**.

The disk identification sector (sector zero) of each disk contains the start sector number and size (in bytes) of the boot file on the disk. In versions of OS-9 earlier than 2.4, to simplify the booting procedure the boot program assumes that the boot file is contiguous. It calculates the number of sectors in the boot file, from the size given in sector zero, and simply reads that many sectors starting at the sector number given in sector zero. Because the boot file size in the identification sector is a 16-bit word, the boot file size is limited to 64k bytes. (This is a historical limitation from OS-9/6809).

From OS-9 version 2.4 onwards, Microware offers the implementor an alternative set of boot ROM example source code, known as 'CBOOT' (because it is written in C). This code has the ability to read any file, using the segmentation information in the file's File Descriptor sector. This uses an alternative form of the information in sector zero. The "boot file size" field is set to zero, indicating that the alternative form is being used. The "boot file start sector number" is the sector number of the File Descriptor sector for the boot file. The boot program reads the File Descriptor sector, and from it takes the segmentation table, which allows it to read a boot file of any size, even if the file is not contiguous on disk.

If boot file sector number in sector zero is zero, the disk has no boot file installed. The **os9gen** utility is used to install a boot file on a disk. It can simply set the values in sector zero to point to a file already on the disk, or alternatively build the file as well, by merging other specified files. If the older contiguous boot file form is used, **os9gen** also checks that the file is contiguous.

The boot file consists simply of modules merged together. The kernel should always be the first module, as many boot programs assume it will be. Modules which cannot fit in the boot file (if the older contiguous form is used, limited to 64k) may be loaded in the 'startup' file. **os9gen** has the following options:

- | | |
|------------------------|--|
| -b=<n> | Set size of memory buffer in which to build boot file - in kbytes. |
| -e | Generate later non-contiguous boot file (can be greater than 64k bytes in size). |
| -q=<path> | Don't build boot file, just set sector zero values to point to existing file on boot disk. |
| -r | Clear sector zero boot file fields - makes disk non-bootable. |

- x Pathlists to files for building the boot file are relative to the current execution directory.
- z[=<path>] Take the list of pathlists to build the boot file from standard input (or a file), rather than from command line parameters.

Use the '-b' option to inform **os9gen** of the expected maximum size of the boot file. The given value must not be less (in kbytes) than the size of the boot file, and cannot be greater than 64 unless the '-e' option is specified.

If the boot program supports non-contiguous boot files, it is much easier to use this facility ('-e' option), rather than being concerned about keeping the size of the boot file below 64k, and ensuring that it is contiguous (although **os9gen** will ensure this if at all possible). In this case the '-q' option of **os9gen** can be used to set the sector zero values to point at any file into which the boot modules have been merged.

If the earlier (contiguous) boot file form is used, **os9gen** must merge the boot file itself, as this gives a much greater likelihood that the file will be contiguous. In this case it is advisable to use the '-z' option, requesting **os9gen** to read the file names to merge from a text file you have created. This makes it much easier to create boot files at a later date, perhaps with modifications for special purposes. If such a file has not been provided with your system, use the **mdir** utility with the '-e' option to display the module directory. The modules in the boot file will be listed first, and there will be a distinct break in the sequence of addresses between the last module in the boot file and the first module loaded after booting.

os9gen and **format** require that the device descriptor for the disk drive has formatting enabled (this is one of the options flags in the device descriptor). Usually this is not the case for hard disk descriptors, to prevent unintentional or unauthorized formatting, and a special descriptor (often **h0fmt** or **fh0**) must be explicitly loaded and used. For example:

```
$ load BOOTOBSJS/h0fmt
$ chd /dd/CMDS/BOOTOBSJS
$ os9gen /h0fmt -z=bootlist -b=100 -e
```

The '-q' option of **os9gen** requires that the device containing the current data directory, or the device name in the pathlist if a full pathlist is given for the file, be the same as the target device name (and it is letter case sensitive). For example:

SYSTEM MANAGEMENT

```
$ load BOOTOBS/h0fmt
$ chd /h0fmt
$ os9gen /h0fmt -q=oldboot
```

5.8 ARCHIVING

Computer data (such as program source files) is usually very valuable, if only because of the time that went into creating it. The loss of some data can have catastrophic effects for a business. Current development projects may be set back by months, and finished products may no longer be supportable. Hard disk drives are by no means infallible, so it is obviously very important to "back up" the computer's hard disk regularly, to minimize the loss if the hard disk does fail.

Yet it is surprising how few development systems are regularly backed up onto tape or other archiving medium. Generally this is because most development systems are not originally specified with a tape drive, and it is often difficult to convince management to buy one as an afterthought. If the computer does not have a tape drive, the only alternative archiving medium usually available is floppy disk, and archiving a hard disk with perhaps 100Mbytes of files onto floppy disks is a tedious and time-consuming task that is generally only undertaken once every few months, if ever.

It is for this reason that a system manager should be appointed for the computer even before it is purchased. He will then have the incentive to ensure that the computer is purchased with a tape drive already installed. If you already have a computer, and it does not have a tape drive (or other high capacity off line storage, such as optical disk), I strongly recommend that you purchase one.

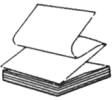
How frequently you back up the hard disk depends on the rate at which you generate valuable data - that is, how long it would take you to regenerate the data created since the last backup if the hard disk fails just before the next backup. Once a week is usually sufficient. Use at least two tapes, and cycle between them, in case the hard disk drive fails while you are generating the backup tape. It is not necessary to save all of the hard disk files to tape at every back up. Instead, you can save only the files that have changed since the last complete back up. This is known as an "incremental" back up. For example, you might do a complete back up every three months, and an incremental back up every week. It is advisable to keep the backup tapes at a different site from the computer system, in case of fire or burglary.

A tape drive is not only useful for backing up the hard disk to protect against hardware failure. Because the tape is removable (unlike most hard disks) it provides "off line storage". That is, any amount of data can be stored, by using additional tapes. To access the data the appropriate tape must be placed in the tape drive. This allows the computer to generate, save, and access unlimited amounts of data, even though it only has immediate rapid access to perhaps 200Mbytes, on the hard disk drive. The saving of data that is not presently required (but may be required in the future), so that space can be freed on the hard disk, is known as archiving. Because the archived data is no longer available on the hard disk it is important to keep a careful written record of what is on each tape. It is also useful to store a printed directory listing with the tape.

Microware provide the **fsave** and **frestore** utilities for backing up (and archiving) and retrieving files. The use of these utilities is described at length in the OS-9 User's Manual. **fsave** and **frestore** will work with any form of storage medium, and provide incremental back up and interactive retrieval facilities.

CHAPTER 6

C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER



This chapter is an overview of the Professional OS-9 program development utilities provided with OS-9 version 2.4 and version 3.2 of the Microware C compiler.

6.1 THE DEVELOPMENT SYSTEM

The development process consists of:

- a) Edit the program source file(s).
- b) Compile and/or assemble to Relocatable Object File(s) (ROFs).
- c) Link ROFs to form a program module.
- d) Test the program, using a debugger.
- e) Repeat the cycle until the program works.

The development tools consist of utilities to facilitate and help manage this process. The development tools provided with Professional OS-9 are:

cc	C compiler executive
cpp	C preprocessor
c68	C compiler (68000 and 68010)

C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER

c68020	C compiler (68020, 68030, and 68040)
o68	assembly language optimizer
r68	assembler (68000 and 68010)
r68020	assembler (68020, 68030 and 68040)
l68	linker
make	automatic compilation manager
debug	assembly-level symbolic debugger

Also available is **sysdbg**, the system state debugger for debugging system state processes, operating system components and multi-tasking applications, and **srcdbg**, the C source level debugger.

The C executive **cc** and **make** are utilities to facilitate the use of the compiler, assembler, and linker, especially in projects with multiple source files.

6.2 THE C COMPILER

The compiler has three phases: preprocessing (**cpp**), compilation (**c68** or **c68020**), and optimization (**o68**).

The preprocessor performs the standard C preprocessing functions – all the lines that start with the '#' character. It produces a temporary file with macros expanded, "include" files inserted, and conditional compilation resolved, ready for compilation.

The compiler translates the C program into assembly language output. (Assembly language is the machine code instruction language for the processor, but in symbolic form). The **cc** executive program has an option ('-a') to halt compilation at this stage. This allows the programmer to see exactly what the compiler has done with his program – particularly important when trying to optimize a critical fragment of the program.

The optimizer looks through the assembly language for common instruction sequences that can be made more efficient. It may change instructions, or even alter the order of instructions.

These phases are not normally called directly by the user. The **cc** executive performs the task of calling each of the phases with appropriate parameters and options, and of creating and deleting all necessary temporary files. **cc** also calls the assembler and linker to produce a finished program module,

normally in the execution directory, ready for testing. **cc** takes a command line option ('-r') to halt compilation after assembly, once a Relocatable Object File (ROF) has been produced (by the assembler). This allows the user to build programs from multiple source files by specifying the linker command line manually, usually within a script file for the **make** utility.

6.3 FILE NAMING CONVENTIONS

cc and **make** use certain conventions for filename extensions. Each extension is a period ('.') followed by a single character. The filename preceding the extension is called the "root". When **cc** and **make** create file names from a given filename, they use the root of the given filename, plus the appropriate extension. A file with no extension in its name is taken to contain an executable program module.

<u>Extension</u>	<u>File contents</u>
.c	C source file
.a	assembly language source file
.r	Relocatable Object File (ROF) – output of assembler
<i>none</i>	executable module – output of linker

The following conventions are also used, although neither **cc** nor **make** recognize them:

<u>Extension</u>	<u>File contents</u>
.h	C definitions source file (used with #include preprocessor directive)
.d	assembly language definitions source file (used with use directive)
.m	assembly language macro definitions source file (used with use directive)
.l	linker library – one or more ROFs in one file

The extension conventions '.c' and '.h' are also recognized by the **umacs** screen editor (it automatically switches on its 'CMODE' mode).

6.4 CC OPTIONS

cc has several options to provide control of the compilation, assembly, and linking phases. Some of the options control which phases will be executed, but most of the options are passed as options to the appropriate phase program. In the following descriptions, the Microware standard notation for command line syntax is used.

<u>Option</u>	<u>Affects</u>	<u>Explanation</u>
-a	cc	Suppress assembly and linking - leave the assembly language in a '.a' file.
-bg	l68	Make the output module a "sticky" module.
-bp	cc	Display phase command lines in full (useful when debugging make files).
-c	c68	Copy C source code as comments to compiler output assembly language file. Use this option with '-a' to help interpret the output of the C compiler.
-d<str>	cpp	Define a symbol - equivalent to #define <str> . Examples: -dFRED #define FRED -dFRED=4 #define FRED 4
-e=<n>	l68	Set output module edition number (default is 1).
-f=<path>	l68	Set output object file name (default is the root part of the source file name for a single file linkage, or 'output' for a multi-file linkage). Output pathlist is relative to the current execution directory.
-fd=<path>	l68	Same as '-f', but output pathlist is relative to the current data directory. Note that the file will not have the execute permissions set.
-g	c68, r68 and l68	Generate symbol table module files with extensions '.dbg' and '.stb' for the symbolic debuggers. These files will go to a directory 'STB' if one exists within the directory to which the output module is directed, otherwise to the same directory as the output module.
-i	cc	Use calls to the cio trap handler for the common I/O functions, rather than including the functions in the program module (significantly reduces the size of small programs).
-j	l68	Do not produce an indirect jump table for long function calls. This suppresses the generation by the linker of an indirect jump table in the program's static storage for function calls where a relative branch offset of more than 16 bits is required. This

<u>Option</u>	<u>Affects</u>	<u>Explanation</u>
		option is not normally used, as a jump table is not created unless needed. Use it if you want to know if a jump table is needed - the linker will report a problem.
-k=[<n>][W L] [CW CL][F]	c68020	<p>Specify processor-dependent compilation. This option enables the compiler generation of extended instruction sequences for 32 bit offsets to static storage and functions, and of 68020/030/040 additional instructions and addressing modes (such programs cannot run on a 68000 or 68010 processor):</p> <p>n=0 for 68000/010.</p> <p>n=2 for 68020/030/040 (allows the use of additional addressing modes and instructions, including long integer division).</p> <p>W indicates that word (16 bit) constant offset indexing be used for data references (default).</p> <p>L indicates that long word (32 bit) constant offset indexing be used for static storage references. Use this option if the program static storage exceeds 64k bytes. It will produce a larger and slightly slower program. For a 68000/010 the compiler will generate two instructions for every static storage reference.</p> <p>CW indicates that word (16 bit) constant offset indexing be used for program references and function calls (default).</p> <p>CL indicates that long word (32 bit) constant offset indexing be used for program references and function calls. Use this option if a program larger than 64k bytes is being generated, <i>and</i> string literals are being referenced at offsets greater than 32k, or the indirect jump table approach is considered slow (each long call takes two instructions through the jump table). This option will produce a larger and (for short calls) slightly slower program. For a 68000/010 the compiler will generate two instructions for every function call or string literal reference.</p> <p>F indicates that in-line FPU instructions be used for maths, rather than function calls. Such programs can only be used with a 68881 or 68882 co-processor, or with a 68040 processor, but floating point maths will be very much faster.</p>
-l=<path>	l68	Specifies an additional library. The library will be searched before any of the default libraries. If multiple '-l' options are used the libraries will be searched in the order of the '-l' options.

C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER

<u>Option</u>	<u>Affects</u>	<u>Explanation</u>
-m=<n>	l68	Add to linker default stack allocation of 3k (units are k bytes).
-n=<name>	l68	Set output object module name - default is same as output object file name.
-o	cc	Exclude optimization phase.
-q	cc	"Quiet" mode - don't display phase execution messages.
-r	cc	Don't execute linker phase - leave output in '.r' Relocatable Object File(s).
-s	c68	Omit stack checking code. The compiler normally generates a call to the subroutine <code>_stkcheck</code> at the beginning of each function, to check for stack overflow. This subroutine is contained in 'cstart.r', and is suitable for programs, but not (for example) for device drivers.
-t=<dir>	cc	Specify directory for temporary files - default is current data directory. This could be a RAM disk ('/r0'), to speed up compilations.
-u<str>	cpp	Undefine a symbol - cancels a preceding '-d' option.
-v=<dir>	cpp	Specify an additional directory to search for #include files. The additional directory is searched before the default ('/dd/DEFS'). If multiple '-v' options are used, the directories are searched in the order of the options.
-w=<dir>	cc	Specify directory for implicit library files - default is '/dd/LIB'.
-x	cc and c68	Use calls to the math trap handler for floating point (and difficult integer) maths, rather than including the maths subroutines in the program.

For example:

```
$ cc -qix test.c
```

The '-i' and '-x' options control which libraries **cc** automatically specifies to the linker (**l68**). These libraries are taken from the directory '/dd/LIB', unless the '-w' option is used to indicate a different directory to search:

<u>Options</u>	<u>Libraries specified</u>
<i>none</i>	clibn.l, math.l, sys.l
x	clib.l, sys.l
i	cio.l, clibn.l, math.l, sys.l
ix	cio.l, clib.l, sys.l

cc automatically specifies 'cstart.r' (also from the default libraries directory) as the first ROF on the command line to the linker. For example, the **cc** command:

```
$ cc -q -bp -ix test.c
```

will (after compilation) produce the linker command line:

```
168 /dd/LIB/cstart.r ctmp.000006 -o=test  
-l=/dd/LIB/cio.1 -l=/dd/LIB/clib.1 -l=/dd/LIB/sys.1
```

'cstart.r' contains the ROF for the startup code for a C program. It is a "root psect" produced from the file 'cstart.a', and must be the first ROF in the linking of a C program. It is *not* appropriate for trap handlers, file managers, device drivers, and other executable modules, for which the programmer must produce (in assembly language) a substitute for 'cstart.a'.

6.5 THE ASSEMBLER

This section describes the Microware assemblers **r68** and **r68020**. It assumes that the reader is already familiar with the 68000 instruction set, and with Motorola-type assemblers. Here the aim is to highlight the special features of the Microware assemblers.

The Microware assembler is a full macro assembler. Two versions are available. **r68** is the standard 68000/010 assembler, while **r68020** supports the additional instructions and addressing modes of the 68020/030/040, plus the coprocessor instructions for the 68881/2 FPU, the 68851 MMU, and the built-in MMUs of the 68030/040.

The assembler contains special functions to help in the production of OS-9 modules, and for use by the C compiler. The syntax of the assembler is Microware's own. The instruction and addressing mode syntax is Motorola standard, but the directives and pseudo-instructions are not compatible with other 68000 assemblers.

The output of the assembler is a Relocatable Object File (ROF). A ROF contains the object code, plus symbolic information, and information required by the linker to allow multiple ROFs to be linked into one object module. In particular, the ROF contains tables identifying code and data offsets within instructions, so that the linker can resolve these at link time into offsets relative to the start of the module and of the static storage. Note that the assembler does *not* produce an output ROF unless the '-o' option is used, to specify the ROF pathlist.

All OS-9 object code is position-independent, and uses address register indirect addressing for data accesses. Therefore the assembler does not provide special functions for the management of absolute-addressed programs.

Symbol names may be of any length. All characters are significant, and letter case is significant. Case is not significant in opcode mnemonics (but it is in user-defined macro names).

6.5.1 The **psect** Directive

The **psect** directive indicates the start of the program code segment of a source file. The **ends** directive indicates the end. Only one **psect** is allowed in a source file. Code-generating instructions are not allowed outside of the program segment. Therefore the **psect** directive is normally one of the first instructions in a source file, and the **ends** directive is usually the last instruction.

The purpose of the **psect** directive is to supply information in the output ROF used by the linker in producing the output module header. The **psect** directive is essentially a pre-defined macro. The syntax of the directive is:

```
psect name,type_lang,att_revs,edition,stacksize,entrypoint,trapentry
```

<u>Parameter</u>	<u>Description</u>
name	psect name - commonly the same as the file name.
type_lang	Output module type and language (word).
att_revs	Output module attributes and revision number (word).
edition	Output module edition number.
stacksize	Stack estimation (zero to use linker default).
entrypoint	Offset to the program execution entry point.
trapentry	Offset to the routine to call for uninitialized trap instructions.

"trapentry" must be omitted if the program does not have a routine to handle uninitialized **TRAP #n** instructions. The offsets to the execution entry point and uninitialized trap instruction handler are relative to the beginning of the psect. At link time the linker adjusts these values to be relative to the start of the module header.

If multiple ROFs are to be linked to form an output module, only one may have a non-zero "type lang" and "att_revs". This is known as the "root psect" (or "non-null psect"). The type, language, attributes, revision, and edition of the root psect determine those of the output module. The C compiler always

produces non-root (null) psects. These are then linked with 'cstart.r', which contains a root psect.

The linker uses the execution offset defined in the first psect on the linker command line as the value to put in the module header execution offset location. Therefore the first ROF on the command line normally contains the root psect. In the case of C programs, 'cstart.r' *must* be the first ROF on the command line, as it provides the initialization function for the C program, which calls the **main()** function.

The following example **psect** uses symbolic names defined in the header file 'DEFS/oskdefs.d' supplied by Microware. This appears to contradict the usual Microware technique of supplying symbolic names in pre-assembled libraries (such as 'LIB/usr.l' and 'LIB/sys.l'). The problem is that the assembler and linker only allow simple addition and subtraction of symbols that are not known at assembly time (external references that will be resolved from a library at link time), and – as can be seen in the example – the "shift left" operator '<<' is frequently used with the **psect** directive.

```
* Program to print a string
      use      /dd/DEFS/oskdefs.d
typelang equ    (Prgrm<<8)+Objct
attrevs  equ    (ReEnt<<8)+0
edition  equ     1
stacksize equ   1000
psect    fred,typelang,attrevs,edition,stacksize,progstart
progstart lea    string(pc),a0    point at string to print
          moveq  #1,d0            print to standard out
          moveq  #strlen,d1       string length
          os9    I$WritLn         print the string
          os9    F$Exit           and exit
string    dc.b    "hello world",13
strlen    equ     *-string
ends
```

6.5.2 The vsect Directive

The **vsect** directive creates static storage segments. Within a vsect, pseudo-instructions are used to reserve static storage and assign symbolic names to static storage locations. Again, the **ends** directive indicates the end of the segment. Any number of vsect segments may appear in a source file, but they must all lie inside the psect. The linker adds up the size of the vsects in multiple ROFs to determine the total static storage required by the program, and to adjust static storage references in program instructions.

The 'ds' directive is used to define uninitialized static storage. The extensions 'b', 'w', and 'l' are used to indicate byte, word, and long word locations respectively. For example:

```
                vsect
fred            ds.l    1           one long word
henry           ds.w    1           one word
jim            ds.b    20          20 bytes
                ends
```

The assembler ensures that word and long word fields are word-aligned (that is, they are on an even address).

The 'dc' directive can be used to define initialized storage, in the same way that it is used to define constant data within a program:

```
                vsect
george          ds.l    2           two long words (not initialized)
percy           dc.l    2           one long word initialized to 2
                ends
```

6.5.3 External Symbols

In a project with multiple source files it is likely that some symbols defined in one source file will be used in one or more other source files, and that symbols defined in libraries will be used in program files. (Note: OS-9 libraries are simply ordinary ROFs merged together).

r68 generates an external reference in the ROF if it encounters a reference to a symbol not defined within the source file.

r68 generates a public declaration in the ROF if a symbol is defined with a terminating colon:

```
fred:          equ      36
```

or

```
henry:         moveq    #0,d0
```

If a symbol is not defined with a terminating colon it is "private" to the source file. It will not appear as a symbol in the ROF, and so cannot conflict with an identical name defined in another source file, even if the other name is defined as public. The C compiler produces public definitions for all objects defined at the outermost scope (functions and static storage), unless the definition is preceded by the **static** keyword, in which case a private definition is generated. Private definitions are generated for all storage defined within functions.

The linker, if requested by the '-g' option, records all public definitions in a separate "symbol table" module, with the extension '.stb'. The symbolic debuggers (**ROMbug**, **debug**, the system state debugger **sysdbg**, and the C source level debugger **srcdbg**) can link to this module to permit the use of symbolic names in debugger commands and expressions (**srcdbg** reads the '.stb' file rather than linking to the module). Symbols defined privately are not known to the debuggers, except to **srcdbg**, which uses the '.dbg' file produced by the C compiler.

The assembler and linker do not permit complex expressions containing external references. Such expressions are limited to adding or subtracting the external symbol. There are also limitations in the use of external symbols in the definitions of other symbols, and of course external symbols cannot be used in conditional assembly statements.

6.6 THE LINKER

The OS-9 linker **l68** is not complex in operation. It takes one or more ROFs and links them to produce an OS-9 object module. One (and only one) ROF must contain a root psect - normally the first ROF. ROFs need not contain object code. For example, they may consist only of public symbol definitions, or static storage definitions. ROFs may be supplied in two ways:

- a) In a file whose name is given as a command line parameter (the file can contain multiple ROFs merged together).
- b) In a library file whose name is given by the '-l' option.

A library is simply one or more ROFs merged together:

```
$ merge rofone.r roftwo.r rofthree.r >mylib.l
```

The linker will include in the output object module all the ROFs specified as command line parameters, plus any ROFs in the libraries required to satisfy external symbol references. ROFs are linked in the order in which they appear on the command line. ROFs in libraries (specified with the '-l' option) are linked after all ROFs not in libraries.

As the linker reads each ROF it attempts to resolve any external references in the ROF from public symbols defined in earlier ROFs. External references that cannot be resolved are added to a table of outstanding references. Therefore once the linker has read all the ROFs specified as command line parameters, it has built a table of outstanding external references that must be satisfied from the ROFs in the libraries.

The linker only scans the libraries once, in the order they are given on the command line. It discards library ROFs that do not satisfy currently outstanding external references. The public symbols defined in a discarded library ROF are also discarded. Therefore it is important to avoid backward references to earlier library ROFs within library ROFs. If a ROF satisfies an outstanding external reference, the whole psect in the ROF (including any vsects within the psect) is added to the output module, and all public symbols defined in the ROF are added to the table of public symbols. The '-l' option of the **rdump** utility can be used to check that a library does not have any backward references within it:

```
$ rdump -l mylib.l
```

rdump will report any backward references within the library 'mylib.l'.

The linker recognizes a special symbol **_sysedit** to set the module edition number, overriding the entry in the root psect. This can be used to set the edition number from within a C source file. The example below uses the '@' character to introduce a single line of assembly language in a C source file:

```
#include <stdio.h>
#include <errno.h>
@_sysedit: equ      3                edition number
```

6.6.1 Linker Options

The linker has several command line options, of which the most important are shown below. Note that the case of the option letter is significant:

- | | |
|-----------|---|
| -a | Generate jump table in static storage for function calls with offsets greater than 32k. |
| -e=<n> | Set output module edition number - overrides edition number in root psect. |
| -g | Output '.stb' symbol module for symbolic debugging. |
| -j | Print jump table information (see '-a'). |
| -l=<path> | Specifies a library file. |
| -m | Print linkage map (values of all public symbols) to standard output. |
| -M=<n> | Specify addition to output module stack size in k bytes. The linker accepts but ignores a negative value. The default stack size is 3k bytes. |

<code>-n=<name></code>	Set output module name (default is name of output file).
<code>-o=<path></code>	Specify output file, relative to the current execution directory.
<code>-O=<path></code>	Specify output file, relative to the current data directory. Note that the file will not have execute permissions set (use the attr utility after linking to set the execute permissions).
<code>-p=<n></code>	Set module header permissions word (in hexadecimal). For example, ' <code>-p=777</code> ' sets read, write and execute permissions for public, group, and owner.
<code>-r[=<n>]</code>	Generate output without module header or CRC, with absolute addressing relative to address <code><n></code> (default 0) in hexadecimal. This option is used to generate boot ROMs, for example.
<code>-s</code>	Print symbol table to standard output.
<code>-S</code>	Make output a sticky module.
<code>-w</code>	Sort printed symbol table alphabetically, rather than by order of value (used with ' <code>-s</code> ').
<code>-z=<path></code>	Get list of ROFs from a file (or from standard input, if no pathlist is given), instead of from the command line.

Example:

```
$ 168 first.r second.r -l=/dd/LIB/sys.1 -o=prog -msw
```

The linker has no interactive features, such as defining symbols at link time.

6.7 THE PROGRAM DEBUGGER

The Microware program debugger **debug** is an assembly-level symbolic debugger. It debugs user-state programs, using the special "debug process" system calls provided by the operating system. The process being debugged exists in its own right, with all the normal facilities and resources of an ordinary process. The difference is that it is not run until the parent (**debug**) makes the appropriate system call, and the parent can install breakpoints (using a system call).

debug automatically attempts to link to or load a symbol table module with the same name as the program module, and the extension '.stb'. It searches first the current execution directory, and then each directory specified in the **PATH** environment variable. For each directory it first searches the 'STB' subdirectory, and then the directory itself. Once the '.stb' file has been found, all publicly declared symbols can be displayed and referenced by name. If **debug** cannot find a '.stb' file for the program it reports an error, but does not abort.

debug also attempts to find a '.stb' file for each trap handler module the program links to. For example, programs generated with the '-i' option to the C compiler use the **cio** trap handler. **debug** reports that no '.stb' file can be found for **cio** when the first **cio** trap call is made by the program.

The debugger provides:

- Program breakpoints.
- Inspection/modification of memory.
- Inspection/modification of processor registers.
- Disassembly of memory.
- Forking the program to be debugged.
- Linking to OS-9 modules.
- Controlled program execution.

The debugger considerably simplifies memory and program inspection by providing an extended set of operators for expressions. For example:

```
dbg: d [.a0+6]+.d0 20
```

means "display \$20 bytes from the address calculated by taking the address stored at the location given by the **a0** register plus 6, and adding the **d0** register". Also, wherever constants are allowed, symbolic references may be used.

Because the 'l' command of **debug** allows linking to any module, **debug** can be used to inspect or modify any module in memory. For example, temporary patches can be made to device descriptors. Such patched modules can be saved to disk (using the **save** utility), and the CRC and header parity can be corrected in the saved file using the **fixmod** utility.

The 'l' command causes **debug** to link to the named module. **debug** puts the address of the module in relocation register 7, known as 'r7'. (**debug** maintains eight relocation registers, which are logically software extensions

to the processor registers). Any relocation register can be named as the default base address to use for display and disassembly, using the '@' command. This is very convenient for inspecting and modifying modules. For example:

```
dbg: l term
dbg: @7
dbg: c 50
0x00000050+r7:18 19
0x00000051+r7:08 .
dbg:
```

This example links to the device descriptor **term** (the usual name for the first serial port), and sets relocation register 7 as the default base address. Location \$50 within the module is inspected and modified. In the case of an **SCF** device descriptor, this has changed the "lines per page" entry from 24 to 25. Of course, this particular operation can be carried out much more simply using the **xmode** utility:

```
$ xmode /term pag=25
```

Note that if the system is using the System Security Module (SSM) for inter-task memory protection, the device descriptor module must have write permission in the module permissions field of its module header. If not, **debug** and **xmode** will generate a bus error (error number 102 - **ES\$BusErr**) when trying to write to the module. By default the linker does not set write permissions when creating a module. The **fixmod** utility can be used to change the permissions of a module in a file. For example:

```
$ fixmod dd.d0 -up=777
```

sets the read, write, and execute permissions for private, group, and public access in the module in the file 'dd.d0'.

debug has two ways of running the program. The first uses kernel-controlled single stepping ("tracing") through the program. The kernel maintains a list of breakpoints for the process being debugged. It executes the program instruction by instruction, until a breakpoint is hit or the program exits. This is slow, but allows breakpoints to be set even if the program is in ROM. The second method ('x -1' command) runs the program at full speed - breakpoints are put into the program code as illegal instructions. When the program hits an illegal instruction the kernel stops the program and wakes up the debugger.

The C source level debugger **srcdbg** uses exactly the same approach as **debug** (except that it reads the '.stb' file, rather than linking to the '.stb' module). In addition, it uses the information in the '.dbg' file generated by the C compiler, assembler, and linker to associate the machine code program

C COMPILER, ASSEMBLER, LINKER, AND DEBUGGER

counter with the C source file. This allows the user to view and step through the program at the C source level, and to view and change C variables. Note that the '.dbg' files are not loadable modules.

CHAPTER 7

THE OS-9 I/O SYSTEM



The OS-9 input/output system has several unique features, making it very flexible and easy to customize. The I/O (input/output) system is tree structured. All I/O system calls go to the kernel, which routes the call to the appropriate file manager module handling that class of device. To perform physical device operations, the file manager calls the device driver module for the specific interface device.

There can be any number of file managers (or none), and for each file manager there can be any number of device drivers. Each file manager handles a particular class of devices. For example, the Random Block File manager (**RBF**) handles randomly accessible block structured devices such as hard and floppy disks, while the Sequential Character File manager (**SCF**) handles character stream devices such as video terminals and printers.

In order to maintain the broad applicability of a file manager, it deals only with logical data operations – it has no understanding of how the data is physically transferred. The physical transfer of data is performed – on the request of the file manager – by the device driver that has been specifically written for a particular I/O interface device, such as a floppy disk controller chip, a serial communications chip, or an intelligent network controller board.

Each device is described by a special data module known as a device descriptor. The device is known by the name of the device descriptor module, preceded by a '/'. For example, the device descriptor module **term** describes the device '/term'. The device descriptor contains the names of the file manager and device driver modules to use to manage the device, essential information about the device – such as the address of the interface – and a set of options fields for controlling the device behaviour.

This approach has two important benefits. It allows the same device driver to be used for any number of I/O interfaces that use the same interface chip, by having a separate device descriptor for each interface, giving a different port address and interrupt vector. Also, multiple device descriptors with different names can be created for the same interface, but with different options settings, or even with a different file manager or device driver name.

For example, two device descriptors could refer to the same serial port, one with options appropriate for its use in communicating with a terminal, another for communicating with a printer. The two device descriptors are "aliases" for the same device. Or, the two device descriptors could specify different device drivers, one for asynchronous communication, another for HDLC synchronous communication. (In the latter example, the device descriptors are not considered simply aliases for the same device, because different device drivers are specified).

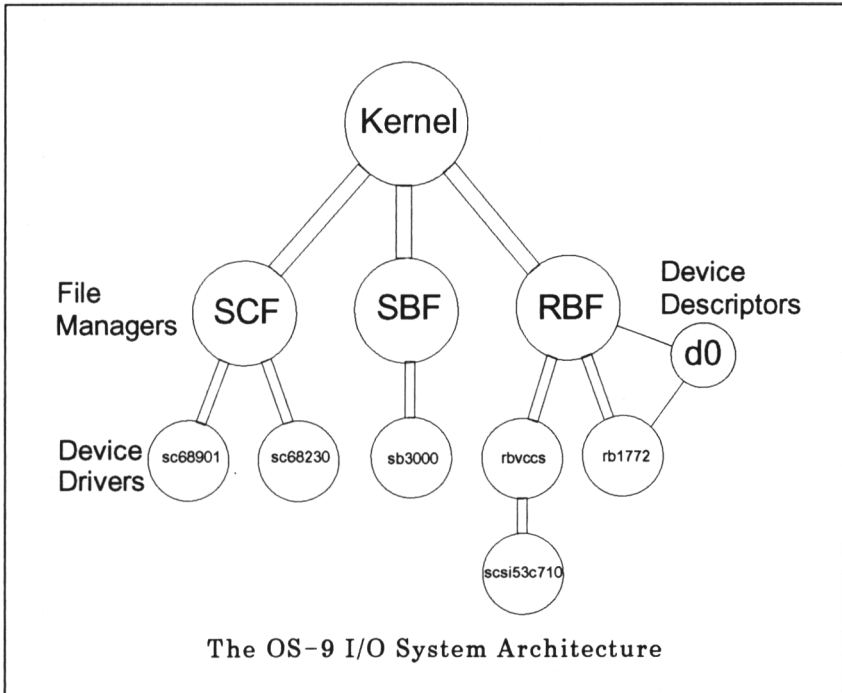
One of the special features of the OS-9 I/O system is the dynamic initialization and termination of I/O sub-systems. Under OS-9, a device does not need to be explicitly initialized by the user. The kernel will automatically initialize the device (if it has not already been initialized) when a path is opened on the device. This feature is described in more detail below.

7.1 I/O SUB-SYSTEMS AND DEVICES

An "I/O sub-system" comprises a file manager module, a device driver module, a device descriptor module, and a device static storage memory area. The I/O sub-system is held together by the device table entry, which contains the addresses of each of these items. (This simple view is made slightly more complex by the possibility of "alias" device descriptors).

A "device" is a physical data store or data conduit. It is difficult to separate the I/O interface on the computer (such as a serial communications chip) from the external data store (such as a disk drive). Some devices have no data store - they exist only as data conduits. Examples are interface chips for serial communications and networking, ADC (analogue to digital converter) chips, and graphics display circuits. Whether or not the device has a data store, it must have an interface on the computer - a microprocessor cannot directly handle any external objects.

The interface is an electronic circuit that appears to the microprocessor as a set of memory locations, and provides the means for the microprocessor to handle the device data, status, and control functions. The interface may be a



simple digital circuit, or one or more specialized chips, or an "intelligent" circuit with its own microprocessor.

The OS-9 concept of a device includes abstract devices that have no interface or data store – they exist only as a function of software. Examples are pipes (sequential memory buffers) and RAM disks (a simulation of a disk drive using computer memory). Programs and the kernel make no distinction in their use of real and abstract devices.

I/O sub-systems are dynamically created and dismantled. When a path is opened on a device, the kernel implicitly executes an **ISAttach** system call. This system call (which is part of the kernel) links to the device descriptor module, and then searches the device table for an entry with the same device descriptor address. If a matching entry is not found, the I/O sub-system does not exist, and must be created. The kernel:

- 1) Gets the file manager and device driver names from the device descriptor.

- 2) Links to the file manager and device driver modules.
- 3) Builds an image of the desired device table entry.
- 4) Sets the device use count in the image to 1.
- 5) Allocates the device static storage.
- 6) Calls the initialization routine of the driver.
- 7) If the driver returned no error, copies the device table entry image to the device table. Otherwise, "detaches" the device (see below).

If a matching entry is found, however, **I\$Attach** performs only steps 1 and 2, and increments the device use count.

Note that the **I\$Attach** system call can also be made explicitly – this is what the **iniz** utility does. This ensures that the device is initialized (if the I/O sub-system did not already exist), and prevents the device from being terminated even if there are no paths open on it. The most common example of the use of this feature is with RAM disks. The RAM disk device driver uses an area of memory to simulate a disk drive, which can be used to store temporary files or copies of commonly required files, to speed up access to these files. Normally this memory is dynamically allocated from system memory when the I/O sub-system is initialized, and de-allocated when the I/O sub-system is terminated. The **iniz** utility is used to ensure that the RAM disk remains in existence even if no paths are open to files on the RAM disk:

```
$ iniz /r0
```

If this were not done, the following sequence of operations would give no error, but an unexpected result:

```
$ copy /h0/startup /r0/startup
$ dir /r0
```

The **copy** would open a path to the RAM disk, causing it to be initialized, copy the file to the RAM disk, and then close the path, causing it to be terminated. The **dir** causes the RAM disk to be initialized again, and reads the root directory of the freshly initialized RAM disk – the file has apparently disappeared!

The complement to the **I\$Attach** system call is the **I\$Detach** system call. The kernel implicitly makes this system call when a path is closed with no remaining duplications (so the path descriptor is about to be de-allocated). An **I\$Detach** system call on an I/O sub-system with a device table use count

of one causes the kernel to dismantle the I/O sub-system and delete the device table entry. The kernel:

- 1) Calls the device driver termination routine (ignoring any returned error).
- 2) De-allocates the device static storage.
- 3) Unlinks from the device descriptor, file manager and device driver.
- 4) Deletes the device table entry.

If the use count was not at one, **I\$Detach** unlinks from the device descriptor, file manager, and device driver, and decrements the device use count.

Note that the **I\$Detach** system call can also be made explicitly – this is what the **deiniz** utility does. This allows the user to terminate an I/O sub-system that is being held in existence even though there are no paths open on it because a previous explicit **I\$Attach** call was made on the device (such as by the **iniz** utility).

A more detailed description of the operation of **I\$Attach** and **I\$Detach** is given below, in the section on The I/O System Calls.

For the system calls **I\$MakDir** (make directory), **I\$ChgDir** (change directory), and **I\$Delete** (delete file) the kernel opens a path, calls the appropriate file manager function, and then closes the path. This is important for two reasons:

- a) All the file manager functions, including those above, are called with an open path (an initialized path descriptor exists).
- b) The device is guaranteed to be initialized when a call is made to any file manager function.

The kernel also increments the device use count before closing the path in the **I\$ChgDir** system call. This is to prevent the I/O sub-system from being terminated if there are no paths currently open on the device on which the default directory resides. Therefore a user wishing to delete an I/O sub-system (that has no paths open on it) must call **deiniz** as many times as **iniz**, **chd**, and **chx** together were called on that device. This is commonly experienced with RAM disks. The **devs** utility can be used to show the

current use count on all devices. (The display from **devs** refers to the use count as "links").

Because the **I\$Attach** and **I\$Detach** system calls can be made explicitly, there may be no path open on the device when the call is made. Therefore the kernel calls the initialization and termination functions of the device driver directly, without calling the file manager. This is to maintain the philosophy that the file manager functions are always called with an open path. The initialization and termination functions are the only device driver functions called by the kernel. All other device driver functions are called only by the file manager.

7.2 FILE MANAGERS AND DEVICE DRIVERS

Microware have separated the code components of an I/O sub-system into a file manager and a device driver. This is a convenience – the split in functionality can be at any level. For example, the pipe device driver does nothing. The operation of pipes cannot vary from system to system, because they are simply memory buffers, so the pipe file manager can contain all the functional code without fear that this will restrict the portability of the I/O sub-system to other computers. Because many device drivers may be written to work with one file manager, the functionality of the device driver and the interface between the file manager and the device driver is a convention defined by the writer of the file manager.

Normally, however, the file manager contains all the code for the logical manipulation of the data for devices of a particular type. For example, **RBF** contains all the functionality for handling a hierarchical filing system. The device driver has only the task of carrying out physical operations on the device (at the request of the file manager).

The separation into two modules has these benefits:

- a) Multiple device drivers (for different devices of a similar type) can use the same file manager, saving on development effort (and memory). Existing file managers can be used wherever possible, independent of the actual hardware on a particular system.
- b) The device driver writer does not have to understand how the filing system works.

- c) The device driver writer has the minimum task to perform – he need only provide low-level physical control of the device.
- d) The device driver writer has a limited (and therefore more easily learned and understood) programming environment.

These advantages simplify and speed up the porting of OS-9 onto new hardware. It should not be thought, however, that the existence of the file manager level makes it impossible for the device driver writer to include special functionality in the device driver. The "get status" and "set status" system calls (described below) can be used to send requests directly to the device driver – the file manager passes on the call without interpretation – so the device driver can implement any number of special features appropriate to the particular device or application.

7.3 DEVICE DESCRIPTORS

Each I/O sub-system is described by a small data module known as a device descriptor. Multiple device descriptors (of different names) may exist to describe the same device, specifying different optional properties or just a different name. Paths to more than one such "alias" can be open simultaneously.

A device is known by the name of its device descriptor module preceded by the '/' character. The "alias" feature of the I/O system means that the same device may be known by more than one name.

A device descriptor contains a standard section and an options section. The standard section follows after the header parity word of the module header. It is defined in the file 'DEFS/module.a'. In the following description, the offsets are relative to the start of the module header:

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$030	M\$Port	1	The device "port address" – the memory address of the registers of the interface chip used to control the device. The kernel only uses this field to check whether a device descriptor is only an alias of another device descriptor already in the device table, and to initialize the V PORT field of the device static storage. The use of this field in actually accessing the chip is a function of the device driver only. This field is intended to allow a device driver module to be used on any number of interfaces that use the same type of chip.

THE OS-9 I/O SYSTEM

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$034	M\$Vector	b	The interrupt vector to be used by the device. This field is not used by the kernel – it is for the use of the device driver when installing an interrupt handler. As with M\$Port , this field is intended to allow a device driver module to be used on any number of interfaces that use the same type of chip. The value set in this field must conform to the requirements of the interface chip. If the chip supports a programmed vector number, this field can be set to a unique number for each chip, so the interrupt handler of the device driver does not need to poll to see which chip interrupted. Note that some chips generate more than one vector (relative to a base value), depending on the cause of the interrupt. In this case, this field should be set with the base value to be programmed into the chip. The chip, or its supporting circuitry, may not support normal vectoring – the hardware is configured to request autovectoring from the processor. In this case, this field must be set to the appropriate autovector – the interrupt level generated by the chip, plus 24.
\$035	M\$IRQLvl	b	The interrupt level of the device. This is usually a hardwired feature of the circuit incorporating the interface chip. Sometimes it is a link option on the board, and some chips or supporting circuitry support a programmable interrupt level. If the interrupt level is hardwired, this field must be set to that value (1 to 7). If it is a link option, this field must match the link setting. If it is a programmable setting or link option, use the philosophy described in the chapter on Device Drivers when deciding on the interrupt level to use.
\$036	M\$Prior	b	The interrupt software polling priority. If the device has been assigned a unique vector number, this field should be zero. The kernel will give an error if a device driver tries to install an interrupt handler on a vector if an entry already exists for that vector and the specified polling priority of the new or existing handler is zero. Some devices or device drivers absolutely require this restriction, because for those devices the vector number returned by the device on interrupt is the only information that distinguishes which device is generating the interrupt. If there is more than one device installed on the same vector the kernel creates a linked list of interrupt table entries in polling priority order (low priority values first). This is the order in which the kernel will call the interrupt handlers on that vector until one of them indicates that it has handled the interrupt.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$037	M\$Mode	b	<p>The device capabilities. This is a byte of bit flags, enabling use of the device as follows:</p> <p>Bit</p> <ul style="list-style-type: none"> 0 read 1 write 2 execute 5 supports "initial file size" 6 supports "non-sharable" files 7 supports directories <p>When a path is opened on a device, the kernel checks that the access mode requested in the open call is compatible with the capabilities of the device.</p>
\$038	M\$FMgr	w	Offset to a string giving the name of the file manager to use.
\$03A	M\$PDev	w	Offset to a string giving the name of the device driver to use.
\$03C	M\$DevCon	w	Offset to an optional table of extra information about the device. If this field is zero, no such table exists. The structure and use of such a table is defined by the device driver writer.
\$046	M\$Opt	w	Size of the options section.
\$048			The options section of the device descriptor.

The options section contains information about the configuration of the device. The structure of the options section is defined by the file manager writer, although specific device drivers may define additional locations to configure special devices (in general, this is not recommended). The options section of the device descriptor (up to a maximum of 128 bytes) is copied to the options section of the path descriptor whenever a path descriptor is created by the kernel.

7.4 PATHS, PATHLISTS, AND FILES

A "path" is a logical conduit for data, commands, and status between a program and a device, or a data structure within a device. A program would not normally access an I/O interface directly (although it is perfectly possible to do so), because this would bypass all the resource management, file handling, and interrupt handling benefits of the operating system. Instead, the program opens a path to the device (or data structure within the device) by using a system call. The operating system returns a path number, which the program uses to identify the path in subsequent operations of read, write,

status, and control. When the program has finished with the device it makes a system call to close the path, closing the logical conduit.

The operating system manages the path through the path descriptor memory structure, which it allocates when a path is opened, and de-allocates when the path is closed. The kernel automatically closes any paths that a process has open when the process dies.

A "file" is a data structure within a device that has a data store. The concept of a file allows one device to be used to store multiple sets of data, and (on most devices) for the data to be modified, extended, or truncated. File management is a function of the particular file manager used with the device, and so will vary between devices. In general, file managers are written to give as similar a programmer's view of files as possible, to make programs more portable.

The best known use of files is on a disk drive. A disk drive is a "random access device" - the computer can read data blocks from all over the disk in any order without a great delay. This makes it feasible to create, modify, extend, truncate, and delete files by allocating space to each file as necessary. To keep track of the files the operating system must maintain one or more "directories" on the disk. A directory is a file that contains a list of file names and positions of the files on the disk. Because a directory is a file, the files in a directory may themselves be directories, creating a so-called tree structure or hierarchy of directories. There must be at least one directory on the disk. This directory is known as the root directory (because it is the root of the tree).

A "pathlist" is a character string identifying a device and/or a file, used in the system call to open a path. A pathlist may have multiple name elements, separated by separator characters. In its simplest form a pathlist is just a device name:

```
/d0
```

or a file name:

```
fred
```

If a device supports files, the device name is taken to refer to the root directory of the device. For example:

```
$ dir /d0
```

will display the root directory of the device '/d0', while:

```
$ list /d0/fred
```

will list the file 'fred' in the root directory of the device '/d0'.

If the pathlist does not start with a device name, it is taken to be relative to the current data directory of the process, or – if the path is opened with the execute mode set – relative to the current execution directory of the process. Because some file managers support directories – which may be hierarchical, or only a root directory – there must be some way of expressing the route through the directory tree to the file required. This is done by stringing the names of the directory files together to make the pathlist, in the order in which the route must be followed.

Under OS-9, the convention is that the name elements are separated by the '/' character, (but bear in mind that this is a function of the file manager, not the kernel). For example, if the device '/d0' has in its root directory a directory called 'GEORGE', and that directory has within it a directory called 'JIM', and within the directory 'JIM' is a file 'henry', then the following command line would be used to list the file 'henry':

```
$ list /d0/GEORGE/JIM/henry
```

Notice that by convention directories are given names in upper case. This is for convenience – it makes it obvious which files are directories when a directory is listed. The kernel uses the system call "parse name" (**F\$PrsNam**) to check device names (and all module names). This call ignores letter case. File managers (including **RBF**) usually also use this system call for the other elements of the path list, so file names are not sensitive to letter case (unlike UNIX). For example, the command line shown above would produce the same result if entered as:

```
$ list /d0/george/jim/henry
```

Again, bear in mind that the kernel only parses the device name, stopping at the first character that is not a legal character within a name. The **F\$PrsNam** system call – used to check module and device names, and the names of files being created – places the following restrictions on names:

- a) Legal characters are numbers, letters, the underscore character '_', the period character '.', and the dollar character '\$'.
- b) The name must contain at least one number, letter, or underscore.

The **F\$CmpNam** system call used to compare the name of a file being opened ignores letter case, and implements wild carding of names. The '*' character matches any number (including zero) of characters, up to the next

occurrence in the target string of the character following the '*' in the match string, or to the end of the target string if the '*' is the last character in the match string. The '?' character matches the next character in the target string whatever it is.

<u>Match string</u>	<u>Matches</u>
f*d	fred, fold, folded, fd
fr*	fred, fr, fritter
f?r	fur, far, for
f*d?d	folded, fielded

7.5 PERMISSIONS, ATTRIBUTES, AND MODES

Devices and files have associated flags to restrict access. These flags are known as "permissions" or "attributes" (there is no difference). When a program opens a path to a device or file, it specifies the "mode" in which it wishes to access the path. The "mode" is a set of flags indicating the type of operations the program wishes to subsequently perform on the path. The operating system checks that the requested mode matches the available permissions of the device or file, and returns an error (such as **ESFNA** - file not accessible) if they do not. The permissions and mode flags are bit flags within a byte or word field. The permissions field may have separate sub-fields for user, group, and public permissions.

When a path is opened the kernel checks the requested mode against the device permissions as part of the **IS\$Attach** system call. The **IS\$Attach** system call checks that all of the mode flags that are set are matched by the equivalent permissions flags in the device descriptor module (except the "sharable" flag, which **IS\$Attach** ignores). It is the responsibility of the file manager to check the requested mode against the individual file permissions. RBF checks that the requested mode is valid for each directory/file in a pathlist. Note that even if a pathlist does not start with a device name, but is relative to the current execution or data directory, the kernel still performs an **IS\$Attach** system call for the device, and so checks the requested mode against the device permissions. If the mode specifies execute access, the kernel checks the device of the current execution directory, otherwise it checks the device of the current data directory.

The basic permission (and mode) flags are read, write, and execute. For example, when used in a permissions field, the "read" flag indicates that read operations are permitted on the device or file. When used in a mode field, the

"read" flag indicates that the program wishes to be able to make system calls to read data from the path. The mode flags – used in a system call to open a path (including creating a new file) – are:

<u>Bit</u>	<u>Meaning when set</u>
0	Read
1	Write
2	Execute
3	Not used
4	Not used
5	Initial file size is specified (when creating a file)
6	Non-sharable
7	Directory file

Bit 5 – used in a call to create a file – is not used by the kernel, and indicates to the file manager that the program is explicitly giving the size of the file to create. Otherwise the initial size depends on the file manager. For example, RBF will create a file of zero length, while the pipe file manager **pipeman** will allocate a pipe of about 90 bytes.

Bit 6 indicates that the calling program wants to be the only process using the file or device. The **ISAttach** system call ignores this flag, but if this flag is set the kernel returns an error when trying to open a path if another path is already open on the device. Notice that the kernel does not perform the reverse check – if this flag is not set, the kernel will allow a path to be opened on a device even if the device has already got another path open on it that was opened with this flag set. Also, the kernel skips this check altogether if the mode or the device permissions include bit 7 – the directory flag.

The permissions flags for a device are held in the **M\$Mode** field of the device descriptor module header. Note that these are the permissions available for opening paths on the devices, as opposed to the module access permissions **M\$Accs** in the module header, which give the permissions available for linking to a module. The device permissions flags have exactly the same format as the mode flags. If a flag – such as the "initial size" flag – is set, this permits the corresponding mode to be used. If the "non-sharable" bit is set, the kernel will not allow a path to be opened on the device if another path is already open on the device, so only one path can be open on the device at any one time.

The permissions flags for files depend on the file manager. RBF keeps a byte field of permissions with each file:

<u>Bit</u>	<u>Meaning when set</u>
0	Owner or group read
1	Owner or group write
2	Owner or group execute
3	Public read
4	Public write
5	Public execute
6	Non-sharable
7	Directory

The use of single flags for owner and group permissions is a historical legacy of OS-9/6809, which does not have the concept of user groups.

7.6 THE I/O SYSTEM CALLS

The I/O system calls are a special subset of the OS-9 system calls. They provide the facilities for data transfer, and control and status of devices and files. While the other system calls all have assembly language symbolic names beginning with the characters **F\$**, the I/O system call names start with the characters **I\$**. The kernel manages I/O calls by using the device static storage, device table, and path descriptor memory structures.

OS-9 has a unified, device independent I/O system. Therefore it has a general purpose set of I/O system calls. It is the job of the file manager to produce an effect in response to an I/O call that is as consistent as possible with the OS-9 I/O system philosophy. Because particular devices usually have some special properties that could not reasonably be covered by a generalized set of system calls, two of the calls - **I\$GetStt** and **I\$SetStt** - are "wild card" calls whose effects vary from file manager to file manager, and device to device. Even with these calls, the device driver writer should try to maintain the same effect for all devices of the same type.

During I/O system calls, when the kernel is making a call to the file manager or device driver it disables the processor data cache(s), unless the compatibility flags in the System Globals indicate that the data caches should not be disabled during I/O accesses (bit 7 of **D_Compact2**), or they indicate

that all the data caches are coherent (**D_SnoopD** is non-zero). When the call to the file manager or device driver is complete, the kernel flushes and enables the data cache(s) it had previously disabled.

In OS-9 version 2.2, after each call to a file manager the kernel would always force a process reschedule by setting the "timed out" flag in the state field of the caller's process descriptor, thus terminating the process's time slice. From OS-9 version 2.3 onwards this is only done if there was another process "I/O queued" on the current process – that is, another process is requesting the resource this process has just finished with. The aim is to maximize I/O usage, as I/O is often the bottleneck in system performance.

For calls made from user state on an already open path (**I\$Seek**, **I\$Read**, **I\$ReadLn**, **I\$Write**, **I\$WritLn**, **I\$GetStt**, **I\$SetStt**, **I\$Close**, and **I\$SGetSt**), the kernel converts the caller's local path number to a system path number through the path number conversion table in the caller's process descriptor. The exception is the call **I\$SGetSt**, as this call is explicitly made with a system path number. Calls made in user state that open a new path return a local path number, and store the system path number in the caller's process descriptor path number conversion table.

Calls made in system state on an already open path expect a system path number, and perform no path number conversion. Similarly, calls that open a new path return a system path number, and do not update the path number conversion table.

The following descriptions of the I/O system calls concentrate on the behaviour of the kernel. Further detail of the behaviour of the **RBF** and **SCF** file managers is given in the section on File Managers.

7.6.1 **I\$Attach**: Add Device to Device Table

The **I\$Attach** system call takes a device name string and a set of mode flags, and ensures that the device is installed in the device table and initialized.

Note that this call is made implicitly by the kernel whenever a path is opened. If the pathlist does not start with a device name, the kernel uses the device table entry address stored in the caller's process descriptor to get the address of the device descriptor on which the current directory is located, and performs an **I\$Attach** on that device (the current execution directory if the mode flags include the execute flag, otherwise the current data directory).

I\$Attach performs the following sequence of operations:

- a) Skip a leading '/' character if present.
- b) Link to the device descriptor module of the given name.
- c) Link to the device driver and file manager modules whose names are in the device descriptor.
- d) Search the device table for an entry for the same device descriptor, or an entry with a different device descriptor specifying the same device driver and port address (an alias). If an entry for the same device descriptor address is found:
 - Check the device static storage address in the existing device table entry. If it is zero, the I/O sub-system is being dismantled – the device driver's terminate routine is currently being executed (it must have gone to sleep).
 - In that case, I/O queue on the process that is terminating the I/O sub-system – its process ID is in the "use count" field of the device table entry. (This prevents a call being made on a device that is in the processing of being terminated.) On wakeup, check again (the device table entry will have been deleted, unless the process was woken for another reason).
- e) If an existing entry for the device descriptor was not found, find an empty entry and build an image of the new entry (in private memory).
 - If the new entry was found to be an alias for an existing entry, copy the address of the device static storage from the existing entry to the image of the new entry.
 - Otherwise, allocate and initialize the device static storage, and call the initialization routine of the device driver.
 - In either case, then copy the image of the device table entry to the new entry, and set the use count in the new entry to one.
- f) If an existing entry was found, increment the use count (unless it is at the limiting value for a word field, 65535).
- g) Check that all the mode flags set in the supplied mode are matched by flags set in the device permissions in the device descriptor. If not, return an error **E\$BMode** (but do not detach the device).

Note that the device descriptor, device driver, and file manager modules are simply linked to. The kernel does not automatically load these modules if

they are not present in the module directory. Therefore the modules must either be in ROM or the boot file, or they must be explicitly loaded before the device is used. The 'startup' file is a convenient place to load additional I/O modules that are regularly required on a particular system.

7.6.2 **I\$Detach: Remove Device from Device Table**

The **I\$Detach** system call is the complement to **I\$Attach**. It is used to remove a device from the device table when the device is no longer in use. The kernel makes this call implicitly whenever it terminates a path – that is, whenever the use count of a path descriptor is decremented to zero, because all duplications of the path have been closed.

I\$Detach performs the following operations:

- a) Decrement the use count of the device table entry.
- b) If the use count is now zero:
 - Get the address of the device static storage from the device table entry, and clear the device static storage field in the device table entry as an indication that the device is being terminated.
 - Look through the device table to see if there is another entry using the same device static storage address. If not, copy the caller's process ID to the "use count" entry of the device table entry, call the termination routine of the device driver, and de-allocate the device static storage.
 - In either case, save the device descriptor address from the device table entry, and clear the device descriptor address and use count fields of the device table entry to zeros, to indicate it is free.
- c) Unlink from the file manager, device driver, and device descriptor modules.

7.6.3 **I\$Dup: Duplicate a Path**

The **I\$Dup** system call takes the path number of an already open path, and returns a new local path number that accesses the same path. The path use count fields (**PD_COUNT** and **PD_CNT**) in the path descriptor are incremented. (The word field **PD_COUNT** is incremented, and the low byte copied to the byte field **PD_CNT**. If the result in **PD_CNT** is zero, it is set to one). In common with the calls that open a new path (**I\$Open** and **I\$Create**), **I\$Dup** uses the first free entry in the process's path number

conversion table. That is, the lowest available local path number is used. This call is used primarily to redirect the standard input, standard output, and standard error paths (paths 0, 1, and 2 respectively) when forking a child.

By duplicating a path, the process can save a copy of its own standard path, close the standard path, open the desired new path – which will take the path number of the closed standard path, being the first free local path number – and fork the child process. The child process inherits the redirected path. The parent can now close the newly opened standard path, duplicate the saved path again – which will be duplicated to the standard path just closed, being the first available – and close the first duplication. **shell** uses this technique for implementing its redirection features.

The kernel uses path duplication when asked to fork a process. It duplicates the requested number of paths (usually three) from the parent to the new child. This is how a child process "inherits" the standard paths of its parent.

Path duplication is a simple function. Owner permissions do not need to be checked, as the process clearly must already have the necessary permissions to have opened the path. Apart from finding a new local path number for the calling process (or the child, in the case of a fork), the kernel simply increments the use count fields of the path descriptor used to manage the path.

7.6.4 **I\$Create: Create a File**

The **I\$Create** system call creates a new file and opens a path to it. File managers that do not support a filing system – such as the Sequential Character File manager (**SCF**) used for character stream devices like terminals and printers – normally treat this just like the **I\$Open** system call. **I\$Create** takes a pathlist giving the name of the new file, a set of permissions flags that determines the permissions of the new file, and a set of mode flags that determines the mode of the path opened to the file.

A directory cannot be created by this call (the "directory" flag of the permissions must not be set). The **I\$MakDir** system call must be used to create a directory.

The kernel treats this call in exactly the same way as an **I\$Open** call. The distinction – creating a new file – is made only by the file manager. **RBF** gives an error if a file of the same name already exists (rather than overwriting the existing file).

7.6.5 I\$Open: Open a Path

The **I\$Open** system call takes a pathlist giving the name of the file or device to open, and a set of mode flags indicating the desired modes of access of the path.

The kernel creates and clears a path descriptor, and allocates a system path number. It initializes the **PD_COUNT** and **PD_CNT** fields of the path descriptor to one, and saves the requested access modes in the field **PD_MOD**. The **PD_USER** field is set to the group and user numbers of the calling process. The kernel then makes an **I\$Attach** system call for the device on which the path is being opened, and saves the device table entry address in the **PD_DEV** field of the path descriptor. If the pathlist does not start with a device name, the kernel makes the **I\$Attach** call for the device whose device table entry address is stored in the "current data directory" field of the process descriptor, unless the "execute" flag is set in the requested access modes, in which case the "current execution directory" entry is used.

If either the requested mode or the device permissions have the non-sharable flag set, and the requested mode does not have the directory flag set, the kernel checks whether a path is already open on the device (using the linked list of path descriptors whose root pointer is in the device static storage). If so, the kernel "detaches" the device, de-allocates the path descriptor, and returns an error **E\$Share** (non-sharable device is in use).

Otherwise, the kernel links the new path descriptor at the head of the linked list of path descriptors open on this device (rooted in the device static storage field **V_Paths**), and copies the options section of the device descriptor to the options section of the path descriptor. This completes the initialization of the path descriptor.

The kernel then calls the file manager. The kernel first checks whether another process is already making a file manager call on the path - the **PD_CPR** (process ID of process using the path) field in the path descriptor is not zero. If so, it "I/O queues" (**F\$IOQu** system call) the calling process onto the process that is currently calling the file manager on this path. This puts the calling process to sleep. When it is woken from the I/O queue, the kernel tries again, unless the process has received a signal (other than the wakeup signal that was used to wake it from the I/O queue), in which case the kernel returns the signal code as an error code. (Of course, in the case of an open or create call the path cannot be in use by another process, as it has just been opened, but this same sequence is used for all calls by the kernel to a file manager).

The process ID of the calling process is then copied to the **PD CPR** field of the path descriptor, indicating that there is currently a call by this process on this path into the file manager, and the kernel calls the appropriate file manager function (in this case the "open" function). On return from the file manager, the kernel "I/O unqueues" the path. It checks whether there is a process I/O queued on the current process (the calling process). If so, it clears the link to that process in the process descriptor of the current process (**P\$IOQN** field), and wakes up that process by sending it a "wakeup" signal (signal code **S\$Wake**). The kernel then sets the "timed out" flag in the process state flags of the process descriptor of the current process, causing reschedule when the current process next returns to user state.

As mentioned above, if the file manager supports directories, opening a path with a pathlist consisting only of the name of the device opens a path to the root directory of the device:

```
path_num = open("/d0", S_IDIR|S_IREAD);
```

RBF implements a special feature that allows a program to open a path to the whole of a disk, as if it were a file. This feature is requested by appending the '@' character to the device name:

```
path_num = open("/d0@", S_IREAD);
```

A process whose group number is zero can read and write any part of the disk in this way. Other processes cannot write to the disk, and can only read the first few sectors (the disk identification sector and the allocation bitmap sectors). Note that a process has a group number of zero if it was forked by a member of the super user group (group zero), or if it has changed its group number to zero using the **F\$SUser** system call (only permitted if the program module was created by a super user).

7.6.6 I\$MakDir: Create a New Directory

The parameters to the **I\$MakDir** system call are the pathlist of the directory to create, the permissions of the new directory file, and the access mode for opening the path while the file is being created. Like the **I\$Create** system call, **I\$MakDir** is a request to the file manager to create a new file, but in this case although the kernel opens a path for the benefit of the file manager, it closes the path before returning to the calling program. The file permissions passed by the calling program are not used by the kernel (although they may be used by the file manager). The "write" and "execute" flags are added to the "read" and "execute" flags of the access modes passed by the calling program, to form the access modes used to open the path.

7.6.7 **I\$ChgDir: Change Current Directory**

The **I\$ChgDir** system call is used to change the current data and/or execution directories. Like the **I\$MakDir** system call, this call temporarily opens a path, calls the appropriate file manager functions, and then closes the path. The parameters are the pathlist of the directory, and the access modes for opening the directory. The kernel adds the "directory" flag to the access modes before opening the path.

If the file manager function is successful, the kernel saves the address of the device table entry for the device on which the directory was opened, in the **P\$DIO** field of the caller's process descriptor. If the access modes have the "read" or "write" flag set, the device table entry address is saved to the "current data directory" portion of this field (the first long word of the first half). If the access modes have the "execute" flag set, the device table entry address is saved to the "current execution directory" portion of this field (the first long word of the second half). Flags of both types may be set, in which case both entries are updated.

Before closing the path, the kernel increments the use count of the device table entry for the device on which the directory exists. This prevents the I/O sub-system being deleted by the **I\$Detach** call used in closing the path, in case there are no other paths open on the device.

7.6.8 **I\$Delete: Delete a File**

The **I\$Delete** system call requests the deletion of a file on a device that supports a filing system. This is another system call that temporarily opens a path, calls the file manager, and closes the path. The parameters are the pathlist of the directory, and the access modes for opening the path. The file manager will also normally insist that the caller has write permission on the file to be deleted. Also, if the file is a directory, a file manager will insist that the directory is empty. In fact, **RBF** does not permit the deletion of a directory. The file attributes must first be changed to make the file an ordinary file, and **RBF** will only permit this if the directory is empty.

7.6.9 **I\$Seek: Set the File Pointer**

The **I\$Seek** system call is made on an open path, and is intended to reposition the current file pointer of a file (that is, the position from which the next read or write will transfer data). The kernel passes this call directly to the file manager.

7.6.10 I\$Read: Read Data

The **I\$Read** system call is intended to read data from a path without editing or interpretation by the file manager. It is made on an open path, with parameters giving the address of the memory buffer to read to, and the (maximum) number of bytes to read. If the call is made from user state, the kernel checks (using the **F\$ChkMem** system call) before calling the file manager that the process has permission to write the requested number of bytes to the indicated memory buffer, and (provided no error is returned from the file manager) adds the number of bytes read to the **P\$RBytes** field of the process descriptor. (If the field thereby exceeds the maximum value that can be stored in a long word - \$FFFFFFFF - the kernel sets it to \$FFFFFFFF).

7.6.11 I\$Write: Write Data

The **I\$Write** system call is intended to write data to a path without editing or interpretation by the file manager. It is made on an open path, with parameters giving the address of the memory buffer to read from, and the (maximum) number of bytes to write. If the call is made from user state, the kernel checks (using the **F\$ChkMem** system call) before calling the file manager that the process has permission to read the requested number of bytes from the indicated memory buffer, and (provided no error is returned from the file manager) adds the number of bytes written to the **P\$WBytes** field of the process descriptor. (If the field thereby exceeds the maximum value that can be stored in a long word - \$FFFFFFFF - the kernel sets it to \$FFFFFFFF).

7.6.12 I\$ReadLn: Read Line

The kernel treats the **I\$ReadLn** system call exactly the same as an **I\$Read** call. However, the intention is that the file manager will end the input when a CR (Carriage Return) control character is read (character code 13), if this occurs before the requested byte count is reached. The file manager may also perform additional data manipulation. For example, **SCF** implements a simple set of line editing functions.

7.6.13 I\$WritLn: Write Line

The kernel treats the **I\$WritLn** system call exactly the same as an **I\$Write** call. However, the intention is that the file manager will end the output when a CR (Carriage Return) control character is written (character code

13), if this occurs before the requested byte count is reached. The file manager may also perform additional data manipulation. For example, **SCF** implements line feed after carriage return, end of line and page pause, and tab expansion.

7.6.14 **I\$GetStt: Get Status**

The **I\$GetStt** "get status" system call is a "wild card" call. In combination with the **I\$SetStt** system call, this call is intended to provide access to all of the features of the I/O system that cannot be accessed by the other calls. **I\$GetStt** is intended to get status about the path, file, or device, while **I\$SetStt** is intended to exercise control or change the state of the path, file, or device. An **I\$GetStt** or **I\$SetStt** call is made on an already open path. In addition to the path number, the caller passes a function code indicating which "get status" or "set status" function is to be executed, together with parameters appropriate to that function.

The kernel implements two "get status" functions itself. After executing the relevant function the kernel also passes the call to the file manager's "get status" routine. Similarly, the file manager will normally pass a "get status" call on to the device driver, even if the file manager has recognized the function code and executed the appropriate function. The kernel or file manager ignores (no error is returned to the caller) an "unknown service request" error (**E\$UnkSvc**) returned by the file manager or device driver respectively in response to a call that it has itself recognized. Any other error is returned to the caller.

If the kernel does not recognize the function code, it passes the call directly to the file manager. Similarly, if the file manager does not recognize the function code, it will normally pass the call directly to the device driver. This allows the file manager writer to invent new function codes for functions specific to the class of devices supported by the file manager, and the device driver writer to invent codes for functions specific to a particular device (or mode of operation of the device).

Microware have defined many function codes, covering all the special functions of the file managers and device drivers they have written. The function codes (which all start with the characters **SS_**) are defined in the file 'DEFS/funcs.a'.

The two "get status" function codes recognized by the kernel are:

<u>Code</u>	<u>Name</u>	<u>Description</u>
\$0000	SS_Opt	Copy the options section of the path descriptor to the caller's buffer.
\$000E	SS_DevNm	Copy the device name (without a leading '/') from the device descriptor to the caller's buffer.

In both functions the kernel checks (using the **F\$ChkMem** system call) that the indicated buffer is permitted to the calling process.

7.6.15 I\$SetStt: Set Status

The **I\$SetStt** "set status" system call is a "wild card" call, complementing the **I\$GetStt** system call. It is intended to allow commands and parameters to be sent to a device and its device driver. The kernel does not implement any "set status" calls itself. It passes the calls directly to the file manager.

7.6.16 I\$Close: Close a Path

The **I\$Close** system call closes an open path. The kernel decrements the **PD_COUNT** use count field of the path descriptor, and copies the low byte to the **PD_CNT** field (if that byte is zero, the kernel copies the high byte of **PD_COUNT** to **PD_CNT**, to ensure that **PD_CNT** only goes to zero if **PD_COUNT** is zero). Provided the **PD_CPR** field is zero, indicating there is not currently a call on the path into the file manager, the kernel calls the "close" function of the file manager. This implies that the file manager is not always called for the closure of every duplication of a path, but it will at least be called for the closure of the last duplication (because there cannot then be any other call currently executing on the path).

If the **PD_COUNT** use count field is now zero, the kernel calls the **I\$Detach** function and de-allocates the path descriptor.

7.6.17 I\$GetSt: Get Status on System Path

The **I\$GetStt** system call, if called from user state, takes a local path number, so a program cannot get the status of paths other than its own. This is a good security measure, but restricts the facilities of programs used to report the status of other processes, such as the **procs** utility.

The **I\$GetSt** system call therefore provides a means for a program to request information about the paths of other processes, by supplying a

system path number rather than a local path number. The calling process must know the system path number for the path it wants information about. It can find this out by requesting a copy of the target process's process descriptor (using the **F\$GPrDsc** system call), and inspecting the process's path number conversion table.

To maintain system security, this system call is restricted to those "get status" functions that the kernel implements itself (get path options, and get device name). The call is only permitted if the calling process is a member of the super user group (group zero), or is the same group and user as the requested path. Furthermore, unlike the **I\$GetStt** system call, **I\$SGetSt** does not normally pass the call on to the file manager.

The kernel implements a special option, using an options field in the extended header of the kernel module. If bit 7 of this field (the byte at offset \$84 from the start of the module header) is set, then the kernel does pass the calls it recognizes on to the file manager after carrying out its own function (and returns no error if the file manager returns the error **E\$UnkSvc**).

7.7 PATH DESCRIPTOR OPTIONS

The second half of the path descriptor is the "options section". The kernel copies the device descriptor options table to the path descriptor options section. The structure of the options in the device descriptor and path descriptor are therefore the same. File managers also commonly write additional information about the path or file at the end of the options section, so that the program can inspect this information using the **SS_Opt** function of the **I\$GetStt** "get status" system call.

The structure of the options section is defined by the file manager writer. Only the first field is common to all options sections. This is a byte field **PD_DTP**, giving the "device type". This is a code number indicating the nature of the device, and the structure of the options section. Its purpose is to allow programs to determine whether they are dealing with an appropriate device, and to determine the structure of the options section. For example, the **tmode** utility checks whether the path uses either the **SCF** or the **GFM** file manager (type 0 or 11 respectively), and gives an error if not. Microware has defined the following device type codes. The symbolic name also indicates which file manager the code is intended for).

THE OS-9 I/O SYSTEM

<u>Code</u>	<u>Name</u>	<u>Description</u>
0	DT_SCF	Sequential character device (terminal or printer).
1	DT_RBF	Random block device (disk drive).
2	DT_Pipe	Pipes.
3	DT_SBF	Sequential block device (tape drive).
4	DT_NFM	Microware protocol network device.
5	DT_CDFM	Compact disc drive (CD-I).
6	DT_UCM	User interface communications device (CD-I).
7	DT_SOCK	Logical socket communications device (ISP).
8	DT_PTTY	Pseudo-keyboard device (ISP).
9	DT_INET	Internet protocol networking device (ISP).
10	DT_NRF	Non-volatile memory store (CD-I).
11	DT_GFM	Graphics display device (CD-I).

A program can alter the fields in the options section (at the discretion of the file manager) by using the **SS_Opt** function of the **I\$SetStt** "set status" system call. In this way a program can dynamically modify the handling of a device. For example, a screen editor will use this mechanism to disable echoing of input characters. The **tmode** utility uses this capability to alter the options on the standard input, output, or error path.

Some options section parameters are used by the file manager, and so the result of changing them is defined in the file manager documentation. Others are used by the device driver, so the effect of changing them may vary from system to system. For example, some serial port device drivers will re-initialize the device if a change is made to the character format or baud rate values in the options section, while others will not. (The recent device drivers from Microware support this feature, but early ones did not).

As mentioned above, the structure of the remainder of the options section depends on the file manager. The options structures for the **RBF**, **SCF** and **SBF** file managers are described below. The offsets shown are relative to the beginning of the path descriptor, and so start at 128. If the symbolic names are used to access the options section of a device descriptor, an adjustment must be applied, because the options section of a device descriptor starts 72 bytes from the start of the module header. The adjustment can conveniently be symbolically expressed as:

```
M$DTyp-PD_OPT
```

For example, to access the baud rate code field of an SCF device descriptor, assuming that the **a1** register contains the address of the device descriptor module header:

```
move.b PD_BAU-PD_OPT+M$DTyp(a1),d0
```

7.7.1 RBF Options Section

Offset	Name	Size	Description
\$080	PD_DTP	b	Device type (1 for RBF).
\$081	PD_DRV	b	Logical drive number (base 0) – used by RBF as an index into the drive tables in the device static storage.
\$082	PD_STP	b	Drive step rate – code depends on the device driver.
\$083	PD_TYP	b	Disk type: <div style="margin-left: 20px;"> Bit Description (when set) 0 <i>before OS-9 version 2.4 – 8" disk (else 5.25")</i> 1:4 Disk size: 1 8" 2 5.25" 3 3.5" 5 Track 0 is double density 6 Removable (hard disks only) 7 Hard disk (else floppy disk) </div>
\$084	PD_DNS	b	Disk density: <div style="margin-left: 20px;"> Bit Description (when set) 0 Double density (MFM) 1 Double track density (96tpi) 2 Quad track density 3 Octal track density </div>
\$086	PD_CYL	w	Number of cylinders available for data (different from PD_TotCyls if partitioning is used, or some cylinders are reserved for defect handling).
\$088	PD_SID	b	Number of surfaces (sides) available for data (tracks per cylinder).
\$089	PD_VFY	b	Verify after write is disabled if this field is not zero.
\$08A	PD_SCT	w	Sectors per track (other than track zero).
\$08C	PD_TOS	w	Sectors on track zero (surface 0 of cylinder 0).
\$08E	PD_SAS	w	Segment allocation size – the minimum number of sectors RBF will allocate when extending a file, to minimize fragmentation.
\$090	PD_ILV	b	Physical sector interleave factor, for formatting.
\$091	PD_TFM	b	DMA mode to use – device driver dependent.
\$092	PD_TOffs	b	First cylinder to use (one for Microware's Universal format, zero otherwise).

THE OS-9 I/O SYSTEM

\$093	PD_S0ffs	b	Lowest physical sector number on each track (zero or one).
\$094	PD_SSsize	w	Logical block size, used by RBF. Prior to OS-9 2.4, RBF only supported a value of 256. Now any power of 2 (starting at 256) is supported.
\$096	PD_Cnt1	w	Options control word: Bit Description (when set) 0 Do not allow formatting 1 Disable multi-sector requests from RBF 2 Device ID will not change 3 Driver supports SS_DSize Get Status call 4 Driver and device can format individual tracks
\$098	PD_Trys	b	(sic) number of retries by driver on data error. 0 => use driver default 1 => one try only (no retries)
\$099	PD_LUN	b	Physical drive number (SCSI LUN).
\$09A	PD_WPC	w	First cylinder to use write precompensation (to disable write precompensation, set this field equal to the number of cylinders).
\$09C	PD_RWR	w	First cylinder to use reduced write current (rarely used).
\$09E	PD_Park	w	Cylinder to park heads on (rarely used).
\$0A0	PD_LSN0ffs	l	Logical sector number offset for driver to add to RBF requests - used to create partitions.
\$0A4	PD_TotCyls	w	Total number of cylinders on disk.
\$0A6	PD_CtrlrID	b	Target controller ID (for SCSI).
\$0A7	PD_Rate	b	Data transfer rate and rotational speed (for floppy disks): Bit Description 0:3 Rotational speed (rpm) 0 300 1 360 2 600 4:7 Data transfer rate (k bits/sec) 0 125 1 250 2 300 3 500 4 1000 5 2000 6 5000

\$0A8	PD_ScsiOpt	1	SCSI options: Bit Description (when set) 0 Host is permitted to assert ATN 1 Driver and interface support target mode 2 Target supports synchronous transfers 3 Check parity on receive
\$0AC	PD_MaxCnt	1	Maximum number of bytes driver and interface can transfer in one request. RBF will not ask to transfer more bytes than this. If there is no limit, set this field to \$FFFFFFF.

The following fields are written by RBF to the path descriptor:

\$0B5	PD_ATT	b	Attributes (permissions) of the file accessed by this path.
\$0B6	PD_FD	1	Logical Sector Number (LSN) of the file descriptor sector of this file.
\$0BA	PD_DFD	1	LSN of the parent directory of this file.
\$0BE	PD_DCP	1	Position of the directory entry for this file in the parent directory file.
\$0C2	PD_DVT	1	Copy of the device table entry address for this device.
\$0C8	PD_SctSiz	1	Copy of the sector size used by RBF on this device.
\$0E0	PD_NAME	b 32	Name of this file (not the full pathlist) as a null-terminated ASCII string (bit 7 of the last character is not set, unlike the name string in the directory entry).

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section – this is a function of the kernel. However, a "set status" call with the same function code only modifies the first 11 fields, up to and including **PD_SAS** (this is a function of RBF).

If the device driver support the **SS_DSize** and **SS_VarSect** Get Status calls to determine the disk and sector sizes, and the device can inform the driver of the relevant values, the following fields can be zero: **PD_CYL**, **PD_SID**, **PD_SCT**, **PD_T0S**, **PD_SSize**, and **PD_TotCyls**.

7.7.2 SCF Options Section

Many of the fields in the **SCF** descriptor options are flags controlling the line editing behaviour of **SCF**. The field description for such flags indicates the behaviour if the field is non-zero. A more detailed description of the behaviour of **SCF** in response to these flags is given in the chapter on File Managers. Several of the other fields are key codes for input editing, pause, flow control, and signal generation. Each feature can be disabled by setting the key code field to zero.

THE OS-9 I/O SYSTEM

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$080	PD_DTP	b	Device type (0 for SCF).
\$081	PD_UPC	b	Flag: force upper case on receive and transmit.
\$082	PD_BSO	b	Flag: to erase a character, transmit [BS][SP][BS], else transmit [BS] (where [BS] is the code in the PD_BSE field).
\$083	PD_DLO	b	Flag: delete line (in response to PD_DEL) by erasing the characters, else start new line (appropriate to teletypes).
\$084	PD_EKO	b	Flag: echo received characters back to device (normal terminal operation).
\$085	PD_ALF	b	Flag: add [LF] after transmitting [CR] ("automatic line feed generation").
\$086	PD_NUL	b	Number of [NUL] characters to send after [CR] (normally zero - set non-zero for slow devices that do not support flow control handshaking, such as teletypes).
\$087	PD_PAU	b	Flag: pause after transmitting a page of lines (number of lines given by PD_PAG) since the last pause or input.
\$088	PD_PAG	b	Length of page in lines, including any top and bottom margins.
\$089	PD_BSP	b	Key code: "delete character" - usually [BS] \$08, sometimes [DEL] \$7F.
\$08A	PD_DEL	b	Key code: "delete line" - usually [^X] \$18. Causes all characters on the current input line to be erased, and the input buffer pointer to be reset.
\$08B	PD_EOR	b	Key code: "end of input line" - usually [CR] \$0D.
\$08C	PD_EOF	b	Key code: "end of file" - usually [ESC] \$1B.
\$08D	PD_RPR	b	Key code: "reprint current input line" - usually [^D] \$04. (Used for devices that cannot erase characters, such as teletypes).
\$08E	PD_DUP	b	Key code: "redisplay to end of line" - usually [^A] \$01. Causes all characters from the current buffer position to the character before the first [CR] character in the buffer to be displayed as if typed in (allows commands to be repeated, with some editing).
\$08F	PD_PSC	b	Key code: "pause at end of next output line" - usually [^W] \$17.
\$090	PD_INT	b	Key code: "generate interrupt signal" (send S\$Intprt to the last process that used the device) - usually [^C] \$03.
\$091	PD_QUT	b	Key code: "generate quit signal" (send S\$Abort to the last process that used the device) - usually [^E] \$05.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$092	PD_BSE	b	Character code used to erase a character (see PD_BSO) – usually [BS] \$08.
\$093	PD_OVF		Character to send on input line buffer full – usually the bell character [BEL] \$07.
\$094	PD_PAR	b	Character format flags (serial communications): Bit Description when set 0 Generate/check parity bit 1 Even parity (else odd) 2:3 Bits per character = 8–field 4:5 Stop bits = field/2 + 1
\$095	PD_BAU	b	Baud rate code: Code Baud rate 0 50 1 75 2 110 3 134.5 4 150 5 300 6 600 7 1200 8 1800 9 2000 10 2400 11 3600 12 4800 13 7200 14 9600 15 19200 16 38400 \$FF use external clock source
\$096	PD_D2P	w	Offset to name string of device for output (echo device). Usually the same as this device (primary device).
\$098	PD_XON	b	Flow control "start" character – usually [^Q] \$11.
\$099	PD_XOFF	b	Flow control "stop" character – usually [^S] \$13.
\$09A	PD_Tab	b	Tab character, recognized and expanded to spaces by SCF during line output (ISWritLn) – usually [^I] \$09.
\$09B	PD_Tabs	b	Tab position spacing (see PD_Tab) – usually 4.
The following fields are written by SCF to the path descriptor:			
\$09C	PD_TBL	l	Copy of the device table entry address for this device.
\$0A0	PD_Col	w	Column number for next character in line output (used for tabbing).
\$0A2	PD_ERR	b	Bit pattern for most recent input character error – format is device driver dependent.

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section – this is a function of the kernel. However, a "set status" call with the same function code only modifies the fields up to and including **PD_Tabs** (this is a function of **SCF**).

7.7.3 SBF Options Sections

The descriptor options structure for the **SBF** file manager is not defined in the file 'DEFS/io.a'. The version of OS-9 supplied at the time of writing only includes the C language header file 'DEFS/sbf.h'. Therefore the symbolic names shown below are those of the C structure 'sbf_opt' in that file. However, Microware has also defined the corresponding assembly language symbols. These are listed in Appendix B.

SBF implements the concept of multiple buffers, so that tape data transfer can continue while the controlling process fills (or empties) the next (or previous) buffer. For systems where the hard disk and the tape drive are on the same interface (typically SCSI), this is usually of no benefit, and the **pd_numblk** field can be set to zero to conserve memory.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$080	pd_dtp	b	Device type (3 for SBF).
\$081	pd_tdrv	b	Logical drive number (base 0) – used by SBF as an index into the drive tables in the device static storage.
\$083	pd_numblk	b	(Maximum) number of buffers to allocate. If this field is zero, SBF buffers are not used (transfer is directly to/from the program's buffer).
\$084	pd_blksize	l	Size of each SBF buffer.
\$088	pd_prior	w	Process priority for the background process that manages continuing transfer using the buffers.
\$08A	pd_flags	w	Device capability flags – the high byte is for use by the file manager, the low byte is for use by the device driver.
\$08C	pd_dmode	w	DMA mode to use – device driver dependent.
\$08E	pd_scslid	b	Target controller ID (for SCSI).
\$08F	pd_scslun	b	Physical drive number (SCSI LUN).
\$090	pd_scsopt	l	SCSI options: <div style="margin-left: 20px;"> Bit Description (when set) 0 Host is permitted to assert ATN 1 Driver and interface support target mode 2 Target supports synchronous transfers 3 Check parity on receive </div>

A "get status" call with function code **SS_Opt** returns a copy of all 128 bytes of the option section – this is a function of the kernel. **SBF** does not modify any fields for a "set status" call with the same function, but returns no error (unless the device driver generates an error other than **E\$UnkSvc**).

7.8 MAKING A NEW DEVICE DESCRIPTOR

Device descriptors are special modules containing data about a particular device. The information is in binary form. The usual way of creating device descriptor modules is by assembling and linking an assembly language file. Microware have provided source files to help the user create new device descriptors. The main work is done by files in the 'IO' and 'DEFS' directories. Which file is used depends on the file manager to be used by the device, because that determines the structure of the options section:

<u>File Manager</u>	<u>File</u>
RBF	IO/rbfdesc.a
SCF	IO/scfdesc.a
SBF	DEFS/sbfdesc.d

These files contain default options section values, which may be overridden. The programmer does not modify these files – they are general purpose. Instead, the programmer creates a separate source file that "includes" the general purpose file at assembly time. Such a source file is created for each device, and normally has the same name as the target device descriptor (with a '.a' extension, being an assembly language source file).

Again, Microware have provided a number of such files (in the 'IO' directory), covering the common device names (for example, 'term.a', 't1.a', 't2.a', 'd0.a', 'd1.a', 'h0.a', 'h0fmt.a'). As with the main descriptor files, the programmer does not normally modify these files (although he may create new ones, using an existing one as a template). Each of these files calls an assembly language macro which the programmer must provide. It is in this macro that the programmer gives the basic information about the device (port address, interrupt vector and level, and device driver name), and overrides the default options values as desired.

For convenience the macros for all the devices in a system are usually contained in one file, called 'systype.d'. This file also usually contains definitions about the system as a whole, such as the memory map of the system. 'systype.d' may be in the 'DEFS' directory, or it may be in a separate

"system" directory from which device descriptors and other operating system components are created for a system. (The latter approach allows multiple target systems to be supported on one development system).

The file 'IO/t1.a' is an example of a descriptor file for an **SCF** device '/t1':

```
nam      T1
ttl      T1 device descriptor module
use      defsfile
use      ../IO/scfdesc.a
T1
ends
```

This file pulls in two other assembly language source files: 'defsfile' (in the same directory as 'systype.d', from where the assembly is performed), and 'IO/scfdesc.a', which does the main work. It also calls the macro 'T1', which must be defined in 'systype.d'. A file for a descriptor for the device '/t2' is almost identical:

```
nam      T2
ttl      T2 device descriptor module
use      defsfile
use      ../IO/scfdesc.a
T2
ends
```

The file 'defsfile' does nothing except pull in two other files:

```
use      ../DEFS/oskdefs.d
use      systype.d
```

Or, if 'systype.d' is in the 'DEFS' directory:

```
use      ../DEFS/oskdefs.d
use      ../DEFS/systype.d
```

The file 'DEFS/oskdefs.d' is supplied by Microware, and includes definitions that cannot be used from a library (due to restrictions of the assembler and linker), such as the module type codes used with the **psect** directive.

The 'T1' macro in 'systype.d' will be similar to this:

```

T1      macro
port    set      $00FFC000
vector  set      27          autovector level 3
IRQLev  set      3
IRQPri  set      5
parity  set      $20          8 bits, no parity, two stop bits
baud    set      14          9600 baud
        SCFDesc port,vector,IRQLev,IRQPri,parity,baud,sc6850
* Default descriptor values can be changed here:
pagpause equ      OFF
DevCon   equ      0          needed from OS-9 Version 2.4 onwards
        endm

```

The macro 'SCFDesc' is defined in the file 'IO/scfdesc.a'. This macro defines symbolic values for use by the main body of the file, from the parameters passed to the macro. The parameters to the macro are the port address, the interrupt vector, the interrupt level, the character format pattern (for the field **PD PAR** in the path descriptor), the baud rate code (for the field **PD BAU**), and the device driver name ('sc6850' in this example).

The file 'scfdesc.a' creates the options section using the 'dc.x' pseudo-operator. For example, the end-of-file field is created by:

```
dc.b    C$EOF
```

The symbolic values used in 'scfdesc.a' are defined in the library 'LIB/sys.l', from the source file 'DEFS/io.a'. If these symbols are not defined within the 'T1' macro the assembler will generate external references for them in the ROF 't1.r', which will be resolved from 'LIB/sys.l' at link time. If one or more symbols (of the correct names!) are defined in 'T1', then the assembler resolves the references at assembly time, and does not generate corresponding external references. So a statement such as:

```
C$EOF    equ      $1A
```

within the macro 'T1' will override the default value (\$1B) for the end-of-file character. The file 'io.a' also defines the symbols 'OFF' and 'ON' (as 0 and 1 respectively), for convenience. For example, the "line feed after carriage return" feature can be disabled by the line:

```
autolf    equ      OFF
```

in the 'T1' macro. Refer to the file 'DEFS/io.a' for a complete list of the symbols used in 'scfdesc.a', and their default values.

The symbol **DevCon** must either be set to zero, or it must be the offset to a table of additional configuration information following the options section. In the source file, this is achieved by placing the table - with the label 'DevCon'

- after the call to the macro 'SCFDesc'. The structure of the additional information is defined by the device driver writer. The above 'T1' macro modified to have such a table might be:

```

T1      macro
port    set      $00FFC000
vector  set      27              autovector level 3
IRQLev  set      3
IRQPri  set      5
parity  set      $20            8 bits, no parity, two stop
bits
baud     set      14            9600 baud
        SCFDesc port,vector,IRQLev,IRQPri,parity,baud,sc6850
* Default descriptor values can be changed here:
pagpause equ      0FF
DevCon   dc.w     $3456
        dc.b     $78,$9A
        endm

```

To make the device descriptor module from the source files:

```

$ r68 ../IO/t1.a -o=RELS/t1.r
$ l68 RELS/t1.r -l=../LIB/sys.l -O=OBSJS/t1

```

If the assembly is done from the 'IO' directory itself, rather than the 'DEFS' directory, or a separate "system" directory, the assembler command line would be:

```

$ r68 t1.a -o=RELS/t1.r

```

If the output file is to go directly to the 'BOOTOBSJS' directory within the execution directory, rather than a local directory, the linker command line would be:

```

$ l68 RELS/t1.r -l=../LIB/sys.l -o=BOOTOBSJS/t1

```

The lowercase '-o' option causes the output from the linker to be relative to the current execution directory, while the upper case '-O' option causes the output to be relative to the current data directory.

Making **RBF** device descriptors using the file 'IO/rbfdesc.a' is similar, but there are subtle differences. 'IO/rbfdesc.a' defines default values locally, rather than producing external references to be resolved from a library. Therefore to change a default value the symbolic definition must be overridden using the **set** pseudo-operator. For example:

```

SctTrk   set      9              sectors per track

```

These redefinitions must follow the call to the 'RBFDesc' macro defined in 'IO/rbfdesc.a', in order to replace the default definitions. Also, one of the parameters to the 'RBFDesc' macro is a conditional assembly symbol, indicating the disk format, or the nearest to the desired format. 'IO/rbfdesc.a'

uses this symbol with conditional assembly to define default options values appropriate to the desired disk format. Refer to the file 'IO/rbdesc.a' for a list of the symbols used for the descriptor fields and for the conditional assembly. For example, a macro to create the floppy disk device descriptor 'd0' might be:

```
DiskD0      macro
port        set      $00FFC040
vector      set      64              first normal vector
IRQLev      set      2
IRQPri      set      0              must be the only interface on this vector
          RBFDesc port,vector,IRQLev,IRQPri,rbteac,dd580
* Default descriptor values can be changed here:
SOFFs       set      1
DevCon      dc.b      "scsi5380",0
endm
```

In this example the device driver to use is **rbteac** and the disk format conditional assembly symbol is 'dd580'. The "first sector on the track" symbol 'SOFFs' is changed from the default value of 0 to a value of 1.

This is an example of a device descriptor for the Microware SCSI Device Driver System, which uses an additional "low-level" (or "physical") driver (actually a subroutine module) to control the SCSI interface, while the "high-level" (or "logical") device drivers interpret the file manager requests and convert them to SCSI commands. This allows multiple devices, even on different file managers, to work through the same interface. The **M\$DevCon** field of the device descriptor is set to the value 'DevCon', which is an offset to the name of the low-level driver module.

A typical device descriptor source file for the device '/d0' would be 'IO/d0.a':

```
nam        D0
ttl        D0 device descriptor module
use        defsfile
use        ../IO/rbdesc.a
DiskD0
ends
```

assembled and linked by:

```
$ r68 ../IO/d0.a -o=RELS/d0.r
$ l68 RELS/d0.r -l=../LIB/sys.1 -O=OBJS/d0
```

It is customary to also produce a "default device" ('/dd') device descriptor for each **RBF** device, using an additional linker command line such as:

```
$ l68 RELS/d0.r -l=../LIB/sys.1 -O=OBJS/dd.d0 -n=dd
```

This will produce a file 'OBJS/dd.d0' containing a module called **dd** - all the other fields will be the same as the device descriptor module **d0**.

The file 'DEFS/sbdesc.d' used to create **SBF** device descriptors is similar to 'IO/rbdesc.a', in that it uses locally defined default values that can be overridden by the **set** pseudo-operator. It does not use conditional assembly to set default groups of values, and so is rather simpler than 'IO/rbdesc.a'. Refer to the file 'DEFS/sbdesc.d' for the symbolic names and default values. A typical **SBF** descriptor macro for a SCSI tape drive (and using no **SBF** buffering) would be:

```
MT0      macro
port      set      $00FFC040
vector    set      64              normal vector
IRQLev    set      2
IRQPri    set      0      must be the only interface on this vector
          SBFDesc port,vector,IRQLev,IRQPri,sbteac
* Default descriptor values can be changed here:
NumBlks   set      0              unbuffered operation
ScsiID     set      3              tape drive SCSI controller ID
DevCon     dc.b     "scsi5380",0
          endm
```

using the source file 'IO/mt0.a':

```
nam      MT0
ttl      MT0 device descriptor module
use      defsfile
use      ../DEFS/sbdesc.d
MT0
ends
```

and assembled and linked by:

```
$ r68 ../IO/mt0.a -o=RELS/mt0.r
$ l68 RELS/mt0.r -l=../LIB/sys.1 -O=OBSJ/mt0
```

7.9 SPECIAL FEATURES

The I/O system has many features that are unique to a particular file manager or device driver. This section highlights a few of the more important special features created by Microware.

7.9.1 RBF Disk Caching

Disk caching uses computer memory to temporarily store data read from disk, or yet to be written to disk, with the aim of speeding up disk file operations. First implemented in OS-9 version 2.4, the disk caching capability in **RBF** is a simple sector-orientated caching without write-behind. Because **RBF** is managing the filing system, the caching functions are able to be somewhat "intelligent", preferentially caching sectors

that are more likely to be needed again. Large block transfers are not cached. The performance benefit of this disk caching varies according to the application, and the allocated cache buffer size. Disk caching is by default disabled, and is enabled using the **diskcache** utility.

7.9.2 SCSI Device Driver System

The Small Computer Systems Interface (SCSI) provides a means of accessing up to 7 controllers through a single interface, with each controller handling up to 8 drives. The drives may be of different types – disk drives, tape drives, printers, and so on. This conflicts somewhat with the simple tree structure of the OS-9 I/O system, as several file managers may be acting through one interface, which must be controlled by one device driver.

The problem is resolved very elegantly using a two level device driver approach. The device drivers known to the kernel and the file managers are "high level" (or "logical") device drivers, each handling one type of controller. That is, they understand the requests from the file manager, and how to build SCSI commands for the controller they have been written for, but they do not know how to transact these commands over the SCSI interface. To do this they call an additional module, known as a "low level" (or "physical") device driver (actually a module of type "subroutine"). The high level drivers link to this module as part of their initialization routine (and unlink from it on termination), so they can call the functions within the low level driver.

SCSI also provides for multiple commands to be transacted concurrently over the interface (known as disconnect/reselect). To implement this feature the low level driver needs its own static storage, in order to keep track of multiple commands together. It does this by means of a data module which it creates in memory when its initialization routine is called from the initialization routine of the high level driver. If the data module already exists (so this is not the first high level driver to be initialized), the low level driver simply links to it. The low level driver returns the address of the data module to the high level driver, which passes this address back to the low level driver when calling the "transact SCSI command" routine of the low level driver.

By dynamically building the name for the data module using the address of the SCSI interface in ASCII hexadecimal, the low level driver allows for multiple SCSI interfaces in the same system.

7.9.3 Ethernet support

The Internet Support Package (ISP) from Microware provides the standard 'TCP/IP', 'telnet', and 'ftp' facilities commonly used over Ethernet networks. ISP uses separate "protocol modules" rather than building the protocol interpretation into the file manager. This allows for the easy addition of new protocols in the future, from Microware or other sources.

7.9.4 The X Window System

The X Window System (often referred to as "X Windows") graphical user interface (GUI) package is also available from Microware. Developed at MIT, X is a very sophisticated package, and is the only GUI available for a wide range of operating systems. It is able to work across a network (such as Ethernet), so that the display terminal can be remote from the computer.

CHAPTER 8

INTER-PROCESS COMMUNICATION

8.1 WHY USE MULTI-TASKING?



Every real time application will have at least two devices to deal with, as the minimum function will be to take in data in some form, process the data, and output the results. Most applications will have more than two devices, perhaps many more. In addition, the application may have some non-real-time devices to deal with, such as an operator keyboard and graphics display.

The direct approach to such an application is to poll each device in turn. If the device needs servicing an appropriate function is called, and then polling continues. However, this approach causes serious problems in most real time applications. The handler function for one device may take some time to execute (for example, the update of a graphical display), so that the real time response of another device is not met.

Again, the direct approach would be polling. A handler function that takes some time to execute can frequently poll the devices that need rapid response, call the appropriate handler if a device needs servicing, and then continue its own function. This produces an increasingly complex program that wastes a great deal of processor time in polling. The tortuous complexity of the program makes maintenance, improvement, and customization of the application very difficult. The processor time wasted in polling forces the use of a much faster processor, or may make the application impossible.

The solution is to use interrupts from external devices rather than polling, and to have a separate program handling each device. The program can sleep, waiting for its device to generate an interrupt, then handle the

interrupt and go back to sleep. Because the operating system will automatically share the processor time between the currently active processes (and give no processor time to the sleeping processes), a program does not have to worry that the time it is taking to service its device might cause an unacceptable delay in servicing another device.

One program – the parent – is called by the operator, or started automatically by the system on startup. It has the job of forking the other programs – the children – that make up the application. Of course, any of the programs may themselves fork other programs. Such secondary programs may execute for as long as the application continues, as the original parent and children normally will, or may be called for a transitory purpose.

But by splitting the application into separate programs, executing as separate processes, another problem has been introduced. A process must be able to exchange data with other processes, and must be able to activate a sleeping process when it has data ready for that process. This is the purpose of inter-process communication.

8.2 WHAT IS INTER-PROCESS COMMUNICATION?

Almost all real time applications require the use of multiple concurrently executing processes. Multiple processes executing together to produce a combined result need mechanisms to:

- Pass data between processes.
- Synchronize one or more processes with each other.

Frequently the two needs are combined – a process requires data from another process, and must be made to wait until the data is available. In addition to processes communicating with each other, interrupt handler functions must be able to communicate with processes. In particular, an interrupt handler must be able to activate (wake up) a sleeping process.

It is also sometimes desirable for a process or interrupt handler to cause a temporary change of flow of control in a program, so that it can handle exceptional circumstances without waiting or polling.

Some inter-process communication is private – that is, between processes who know of each other's existence and wish to communicate directly between themselves. Other communications are public – the sending process is essentially broadcasting, and some or all other processes can receive the communication.

Because an interrupt can occur while a system call is being executed, the operating system must mask all interrupts during system calls that can be called from an interrupt handler (to prevent concurrent use of the same operating system memory structures). To avoid masking interrupts unnecessarily (because it would adversely affect real time response), OS-9 specifies that only certain system calls are allowed in an interrupt handler. Therefore not all of the inter-process communication mechanisms can be used by an interrupt handler communicating with a process.

8.3 OS-9 INTER-PROCESS COMMUNICATION FACILITIES

Multiple inter-process communication mechanisms are required to efficiently service the various circumstances that require data transfer or inter-process synchronization. OS-9 provides several inter-process communication mechanisms. Deciding which to use to solve a particular problem is part of the application design task. The table in figure 6 highlights the important differences between the mechanisms. The columns of the table are headed as follows:

- DAT Data can be passed.
- SYN The mechanism provides synchronization (it can wake up a sleeping process).
- PUB The mechanism is public, as opposed to private.
- INT An interrupt handler function can use this mechanism.

Mechanism	DAT	SYN	PUB	INT
Wait for child	Exit status	Yes	No	No
Signal	Signal code	Yes	No	Yes
Event	Event value	Yes	Yes	Yes
Alarm	No	Yes	No	Yes ¹
Unnamed pipe	Yes	Yes	No	No
Named pipe	Yes	Yes	Yes	No
Disk file	Yes	Yes	Yes	No
Data module	Yes	No	Yes	Yes
Shared memory	Yes	No	Yes	Yes
External memory	Yes	No	Yes	Yes

¹Generated by clock tick interrupt.

• **Figure 6. Inter-process communication mechanisms**

8.4 FORKING A PROCESS

The forking of a child process may not seem like a form of inter-process communication, but in fact it is the most fundamental form of inter-process synchronization and communication. Unless additional processes are forked no inter-process communication can take place (almost by definition!). Forking provides synchronization (the forked process starts when the fork request is made) and communication (the parameter string passed to the child). Lastly, waiting for a child to die is a basic form of synchronization and communication (by the exit status of the child).

This form of inter-process communication is used every time a command line is entered through **shell**. **shell** forks the requested program, and then waits for it (or any other of its child processes) to die. When it dies, **shell** reports its exit status to the user if it is not zero. **shell** provides variations on the use of this basic mechanism: concurrently executing processes ('&' and '!'), implicit forking of another shell (parentheses), and waiting for one or all children to die ('w' and 'wait').

A process is forked using the **F\$Fork** system call. The parent process specifies the name of the program module, or the name of the file from which the program module is to be read (relative to the parent process's current execution directory). The parent also passes a pointer to a parameter string and the length of the string, the number of paths the child should inherit from the parent (usually three), and (optionally) an additional static storage size and process priority. The kernel adds the parameter string length to the data storage size specified in the program module header, the stack size in the program module header, and the additional static storage specified in the fork request, to make the total static storage size to allocate for the child. The kernel copies the parameter string to the top of the child process's static storage.

In effect, the parameter string is a message passed from the parent to the child process. The parameter string can be any byte string, although an ASCII text string or sequence of strings is usually used. The Microware C library functions **os9fork()** and **os9forkc()** allow the **F\$Fork** system call to be made from C. The difference is that **os9forkc()** allows the programmer to specify the number of paths to inherit, while **os9fork()** implicitly asks for three paths to be inherited. However, the library also provides a higher level function - **os9exec()** - that converts an array of text strings together with the environment parameter strings of the parent into a single parameter string compatible with **F\$Fork**. The 'cstart' function, which is the startup

function of a compiled C program, converts this string back into an argument array and environment parameter array, as expected by a C program.

The forked process is immediately put in the active queue, and so will be allocated processor time in its turn. Therefore the parent cannot assume that it can execute further instructions between the fork request and the child's first slice of processor time. It is quite possible that the parent process will finish its time slice during the fork system call, and that the new child process will start execution immediately the call is finished.

Below is an example forking of a child program, specifying three paths (0, 1, and 2) to be inherited and no additional static storage memory. The process priority value of zero specifies that the child process should have the same process priority as the parent. Note that by convention (for C programs) the first argument string is the name of the program, although this has no effect on the **F\$Fork** system call or the **os9exec()** library function.

```
char *args[]={
    "tmode",          /* program name */
    "nopause",        /* argument strings */
    "noecho",
    NULL              /* NULL pointer terminates the list */
};
int child_pid,        /* child process ID */
    dead_id,          /* process ID of dead child */
    status;           /* child exit status */

int os9fork();
extern char **environ; /* environment parameter array */

/* Fork the child (could be one of many): */
void fork_child()
{
    child_pid=os9exec(os9fork,args[0],args,environ,0,0);
    printf("Forked process ID %d\n",child_pid);
    /* Wait for child (could be any child) to die: */
    dead_id=wait(&status);
    printf("Process ID %d died with exit status %d\n",dead_id,status);
}
```

The **F\$Wait** system call, called by the **wait()** C library function, waits for any process that is a child of this process to die. The system call returns the process ID of the dead child, and its exit status code. A non-zero exit status is usually considered an error, but as this is only interpreted by the parent it can be used instead to return a result value. Note that the parent cannot specify which child it is waiting for, and that the parent can be woken instead by a signal, in which case the reported child ID and exit status are zero. Therefore if the process has forked more than one child it will need to

check which child has died, and if it has installed a signal intercept handler (see the section on Signals) it will need to check whether it was woken by the death of a child or by a signal.

A process cannot "miss" the death of a child, even if it is not waiting when the child dies, or more than one child dies simultaneously. Although all of the resources of the child are de-allocated by the kernel when the child dies, the child's process descriptor is retained until the parent process executes an **F\$Wait** system call and is returned the exit status of the child, or the parent process itself has died. This guarantees that this form of inter-process communication cannot fail.

The **F\$Fork** system call can be used from system state. For example, a device driver can fork a process. In this case the parent of the new process is the current process, that is, the process that made the system call as part of which the child was forked. As the device may later be used by other processes it may be desirable to disinherit the process (make it an orphan) by cutting the links to the parent process. This is described in the chapter on Multi-tasking.

8.5 SIGNALS

An OS-9 signal is a small transitory message sent from one process to another. It can be viewed as the software equivalent of a one word telephone call - it wakes you up if you were asleep, interrupts what you were doing, and gives you a small piece of information as to the reason for the call. It is very important to distinguish between signals and interrupts (a common source of confusion). A signal is purely a software mechanism, while an interrupt is a hard-wired response of the microprocessor to an external electrical signal. The confusion arises because, under OS-9, both cause an asynchronous change of flow of control of software, and both can be "masked". However, there the similarity ends, and the two mechanisms are completely separate.

The **F\$Send** system call is used to send a signal from one process to another, or to all processes of that user (a broadcast signal). It requires knowledge of the process ID of the destination process, unless the signal is broadcast. The sending process must have the same user ID and group number as the receiving process, or be a super user (group zero). Sending a signal has two important effects on the destination process:

- a) The process is made active if it is not already active, and so will become the current process at some time in the future (it may already be the current process).
- b) The process's signal handler function is called when the process next runs in user state.

It is important to bear in mind that at the moment the signal is sent – either from another process or an interrupt handler – the destination process cannot be executing in user state, even if it is active, and even if it is the current process. Even if the process sends a signal to itself, the sending of the signal is carried out in system state by the **F\$Send** system call. Therefore once the signal has been sent, the process's signal handler function will be the next part of the program to execute (after any system call the program is making finishes), even if the program was active and only part of the way through a subroutine.

Under OS-9, the signal mechanism is the only inter-process communication mechanism that can cause such an asynchronous change of flow of control. Also, signals are the only way of waiting for multiple sources of synchronization without polling, because a signal forces a process to become active. Once the signal handler function finishes, execution of the program continues as before. If the process was sleeping or waiting, execution continues with the instruction following the "sleep" or "wait" system call.

If the destination process was executing a system call, the signal handler function of the process is not called until the system call finishes (the process returns to executing in user state). If the process was sleeping within a system call (for example, in a device driver, waiting for an interrupt), the process is woken and continues execution after the sleep system call. A signal makes a process active irrespective of its previous state (and is the only way of waking a process from an untimed sleep). Therefore a signal will wake a process that is, for example, waiting for an event, or waiting for a child to die. So, on return from such "waiting" system calls it is important to check the reason for the return from the system call – did the system call finish for the intended reason, or was a signal received?

When a process is about to continue execution in user state (after completion of a system call or an interrupt) the kernel checks whether a signal is pending for the process. If a signal is pending, the kernel checks whether the process has installed a signal handler function. If so, the kernel builds an additional stack frame on the process's stack so that execution will be diverted to the signal handler function, exactly as if the program had made a

subroutine call to the signal handler. On return from the signal handler the program continues execution as before.

A process may receive multiple signals before becoming the current process. The signals are queued (within the recipient's process descriptor) in the order in which they have been received. The **P\$Signal** field of the process descriptor contains the latest signal code received¹, or zero if no signals are pending. A signal handler function should end with an **F\$RTE** system call, rather than a "return from subroutine" instruction. This system call does very little, but its exit causes the kernel to check again whether a signal is pending for the process, and call the signal handler function again if so. In this way all pending signals are handled before normal program execution recommences. Note: the "wakeup" signal **S\$Wake** and the "kill" signal **S\$Kill** are not queued – see below.

A process installs (or cancels) a signal handler function using the **F\$Icpt** system call. If a program intends to handle signals, installing the signal handler function should be one of the first instructions in the program. The process will be aborted if it does not have a signal handler function installed when the process is about to execute in user state and a signal is pending (except for the "wakeup" signal **S\$Wake** – see below). The "kill" signal **S\$Kill** cannot be handled – it always aborts the receiving process.

The signal intercept routine is passed the signal code (in the low word of the **d1** register), and the number of signals queued including the current one (in the low word of the **d0** register). It is not passed the process ID of the sender.

The C library function **intercept()** is used to install a signal handler function in a C program. The signal handler is a normal C function. The **intercept()** function ensures that the **F\$RTE** system call is made on exit from the signal handler function. Passing a null pointer (zero) instead of a function address cancels any currently installed signal handler for the process.

The signal codes 2 to 31 are known as the "deadly" signals². Device drivers that are sleeping (waiting for completion of a device operation) will normally abort their operation and return to the caller with an error if woken by such a signal, unless this might be destructive to a filing system. Signals 2 to 32 can be ignored, by setting the corresponding bit in the **P\$SigMask** field in the process descriptor (signal 32 is ignored by setting bit zero). This function

¹ Before OS-9 version 2.4, the **P\$Signal** field contained a copy of the first signal in the queue – the oldest pending signal – or zero if no signals were pending.

² Before OS-9 version 2.4, only signal codes 2 and 3 were considered deadly.

is not available through a standard system call. Ignored signals are not queued, and do not cause the receiving process to become active.

Signal code 0 - **S\$Kill** - is the "kill" signal. It is not queued. Instead, the kernel sets the "condemned" flag in the **P\$State** field of the process descriptor. When the process is about to restart execution in user state it is unconditionally terminated. The name "kill" can cause confusion to C programmers, as the C library function **kill()** is the C function for sending signals. The C function is called **kill()** for compatibility with the UNIX standard C library, but it is used to send any desired signal code.

Signal code 1 - **S\$Wake** - is the "wakeup" signal. It is specifically intended for use by an interrupt handler to wake up a sleeping device driver (or other operating system component). It should not be used to signal a user state program, as its special properties do not guarantee proper inter-process synchronization. This signal is not queued, and so does not cause the recipient's signal handler function to be called.

8.5.1 Masking Signals

In order to guarantee correct inter-process synchronization, avoiding the possibility of a timing "race" condition, it is important to be able to "mask" signals, in the same way that interrupts can be masked. That is, the response to the signal can be held off until the process is ready to respond. Signals - except the "kill" signal - can be masked using the system call **F\$SigMask**.

This system call increments, decrements, or clears the signal mask field **P\$SigLvl** in the caller's process descriptor. While this field is non-zero signals sent to the process are queued in the recipient's process descriptor (except the "kill" and "wakeup" signals) and force the process into the active queue as normal, but the process's signal handler function is not called until signals are unmasked.

The **F\$SigMask** system call takes a single parameter, which must be -1, 0, or 1. If the parameter is zero the field **P\$SigLvl** in the process descriptor is cleared (signals are unmasked). If the parameter is one **P\$SigLvl** is incremented (unless it is already 255). If the parameter is minus one **P\$SigLvl** is decremented (unless it is already zero). This approach allows calls to mask signals to be nested. The kernel masks signals (increments **P\$SigLvl**) for the process before calling the process's signal handler function (the signal mask must have been clear for the kernel to decide to call the signal handler), and unmasks signals (decrements **P\$SigLvl**) on return from

the signal handler function, as part of the **F\$RTE** system call. This ensures that the signal handler function will not be called recursively.

If the signal handler function increments the signal mask, the mask will not return to zero when the kernel decrements it, so any other pending signals are not serviced until the program clears the signal mask. This permits the main body of the program to respond to signals one at a time³.

Executing the system call **F\$Sleep** (timed or indefinite sleep) or **F\$Wait** (wait for child to die) from user state clears the **P\$SigLvl** signal mask. Therefore the following sequence works correctly as a user state inter-process synchronization mechanism, without the risk of a "race" condition:

- 1) Mask signals using **F\$SigMask**.
- 2) Check a flag set by the signal handler function indicating that a signal has been received.
- 3) Sleep using **F\$Sleep**.

An example in C is shown below.

```
int got10,          /* flag - signal 10 received */
    got_20,         /* flag - signal 20 received */
    invalid;        /* flag - invalid signal received */

sighandler(s)      /* the signal intercept handler */
register int s;     /* signal code received */
{
    switch (s) {    /* record which signal has arrived */
        case 10:
            got_10=TRUE;
            break;
        case 20:
            got_20=TRUE;
            break;
        default:
            invalid=s;
            break;
    }
}
```

³ This facility was not available prior to OS-9 version 2.3 - the **F\$RTE** system call called the program's signal handler function again if another signal was pending, without checking the signal mask.

```

main()
{
    while (TRUE) {
        sigmask(1);           /* mask signals */
        if (got_10 || got_20 || invalid) {
            sigmask(0);       /* unmask signals */
            if (got_10)
                printf("Received signal 10\n");
            else if (got_20)
                printf("Received signal 20\n");
            else exit(_errmsg(1,"Invalid signal %d received\n",
                invalid));
        }
        else
            tsleep(0);         /* sleep until woken */
    }
}

```

In the above example, signals are masked before checking whether a signal has already arrived. Once signals are masked the signal handler function will not be called even if a signal arrives, so the flags cannot be set between making the check and going to sleep. Calling the **F\$Sleep** system call (via the **tsleep()** or **sleep()** C library functions) unmask signals⁴ and checks for any signal pending. If a signal is pending the process is not suspended – the system call returns immediately. Otherwise the process is put in the sleeping queue. The kernel performs these actions with interrupts masked, so even if the signal is to come from an interrupt handler function these steps are indivisible – a signal cannot arrive between checking for signals and going to sleep.

This check for a pending signal is not just a check of whether the **P\$Signal** field (most recently received signal code) of the process descriptor is not zero. If this were done, a device driver, or other operating system component, woken from a sleep by a signal being used for inter-process communication, would not be able to go back to sleep again (waiting for a signal from an interrupt service routine), because the pending signal would cause the **F\$Sleep** system call to return immediately. Therefore the kernel sets a flag in the process descriptor (bit 7 of the **P\$SigFlg** field) whenever a signal is received when the process is active. User state calls to "sleep", "wait for child", or "wait for event", clear this flag, but calls made in system state do not. Before suspending the process these calls check whether this flag is set. If so, the flag is cleared, but the process is not suspended – the system call returns immediately. In effect, in system state this flag indicates that the process has received a signal since the last "sleep", "wait for child", or "wait for event" system call. Thus a system state function is only woken once by

⁴ Unless the system call is made from system state.

each signal, and can go back to sleep even though a signal is pending in the process descriptor.

If the "sleep", "wait for child", or "wait for event" system call is made from user state, the system call clears bit 7 of the **P\$SigFlg** field immediately. However, it then checks whether there is a signal pending (the **P\$Signal** field of the process descriptor is not zero). If so, it sets the bit. This is detected by the main body of the system call routine (common to calls from system state and user state), which – as described above – returns immediately to the caller. Thus any pending signal causes the process to continue execution. Note, however, that the "wakeup" signal (**S\$Wake** – code 1) is not queued, nor is it put in the **P\$Signal** field – it just causes bit 7 of the **P\$SigFlg** field to be set. Therefore a call to "sleep", "wait for child", or "wait for event" made from user state cannot detect this signal. The process will be suspended even if this signal was received since the last "sleep" call, unless the call is made in system state and there has been no intervening call made from user state. Thus the "wakeup" signal is not suitable for inter-process communication – it is intended only to be used by an interrupt service routine waking up an operating system component.

This somewhat complex distinction between the effect of these calls in user and system state reflects the fact that to the user state program these are inter-process communication mechanisms, but to an operating system component (such as a device driver) they are mechanisms for communication between an interrupt service routine and a process.

As described above, the **F\$Wait** system call performs the same operations as the **F\$Sleep** system call with regard to pending signals, but note that prior to OS-9 version 2.3 the **F\$Wait** system call tested for a pending signal by checking the **P\$Signal** field directly, rather than using bit 7 of the **P\$SigFlg** field, so it was not suitable for use from system state.

Note that the "wait for event" system call does *not* clear the signal mask, but it does check for a signal pending⁵. If a signal is pending, the system call returns immediately to the caller (in which case the returned event value is the current event value). Note also that if a process waiting for an event receives a signal while it has signals masked (the **P\$SigLvl** field is not zero), the process is still forced active, but the signal handler function will not be called – the process continues execution with the instruction following the "wait for event" system call. This means that if a process makes a "wait for event" system call with signals masked, but a signal is pending or is received before the event changes to the desired range, the "wait for event" terminates

⁵ From OS-9 version 2.3 onwards.

and the process is returned the event value at the time it made the "wait for event" call (which will be outside the desired range), but the program's signal handler function is not called (until signals are unmasked).

Before OS-9 version 2.3 there was no C library function for the **F\$SigMask** system call, so one is given in assembly language below:

```
#asm
sigmask:  move.l  d0,d1          copy mask value (0, 1, or -1)
          moveq   #0,d0         d0 must be zero
          os9     F$SigMask     execute the system call
          rts
#endasm
```

8.5.2 Signals – Cautions

This section covers particular details of the operation of signals that are most commonly the source of problems when using signals.

The "wakeup" signal **S\$Wake** is not queued. If it is received while the process is in system state it is lost on return to user state, and so is not suitable for inter-process synchronization. It is primarily intended as a mechanism by which an interrupt handler function can wake up a sleeping device driver (or other system state component).

The **F\$Sleep** and **F\$Wait** system calls return to the calling process immediately if a signal is pending. In system state, a pending signal will only cause a process to wake up once – a further signal must be received to wake the process from a subsequent "sleep" system call made before returning to user state.

The signal intercept handler function is called when a process with a signal pending is about to return to user state (it becomes the current process, or – if it is already the current process – it returns from a system call or an interrupt service routine finishes). Therefore system state processes cannot make use of a signal intercept handler function.

8.6 EVENTS

OS-9 events are another mechanism for inter-process synchronization. OS-9 events are similar to signals in a number of ways:

- a) An event is used for inter-process synchronization.
- b) An event passes a small amount of data – the event value.

- c) Events can be used from interrupt handler functions.

However, events differ from signals in some important respects:

- a) An event cannot cause a temporary change of flow of control – there is no equivalent to the signal handler function.
- b) A process cannot be woken by an event when waiting for something else – a process can only be woken by an event when waiting for that event to change.
- c) Events are public – any number of processes can link to an event, and alter the event value or wait for it to change.
- d) Events are more flexible than signals – the event value can be changed in a number of ways, and a process can wait for the event value to change to within a desired range.
- e) Events are not transitory – an event exists even when not being changed or waited for, and its current value can be read.

To use an event it must first be created. A system call is used to create an event, giving a name (character string) for the event, and an initial value for the event. The name can be up to 12 characters, and is subject to the same restrictions as module names. Letter case is not significant. The kernel finds a free entry in the event table (checking that no event of the same name already exists), and initializes the entry.

The event is allocated an event ID. This is a long word, of which the high word is the event number from the high word of the **D_EvID** field of the System Globals (after it has been incremented), and the low word is the index (base zero) of the event entry within the event table. Before creating the event ID the kernel increments the high word of the **D_EvID** field. The kernel does not permit the high word of the **D_EvID** field to be zero (if it becomes zero it is set to one), so the event ID cannot be zero. The event table is dynamically extendible, so the number of events in existence at any one time is not limited. The caller creating the event is returned the event ID (or an error if an event of the same name already exists).

The event table entry also contains a link count (initialized to one when the event is created), a "signal increment" (also known as an "automatic increment"), and a "wakeup increment". The event value is a signed long word and the increment values are signed words. The increment values are specified when the event is created. The "signal increment" (nothing to do

with OS-9 signals) is added to the event value when the "signal event" call is made. This is a convenience, being simpler to use than an explicit change to the event value. The "wakeup increment" is added to the event value when a process is woken from waiting on an event because of a change to the event value.

Once an event has been created, other processes can get the event ID by making a "link to event" call, specifying the event name. Each such call increments the event link count, in a similar manner to the link count of an OS-9 module directory entry. Similarly, once a program has finished with an event it makes an "unlink from event" call, which decrements the link count for the event. Once the link count has been reduced to zero the event can be deleted by a "delete event" call, which frees the event table entry. Because an event could be unlinked more than once by the same process, reducing its link count to zero even though other processes are waiting on the event, the kernel will wake up any processes waiting on an event that is being deleted (returning them an "invalid event ID" error - **E\$EvtID**). Note that when a process is terminated the kernel does not automatically unlink or delete any events a process may have linked to or created.

A process that has the event ID can make use of the event in three ways:

- a) Read the current event value.
- b) Change the event value.
- c) Wait for the event value to fall within a specified range.

Whenever the event value is changed, the kernel function that makes the change checks whether the new event value falls within the specified range of any process waiting on the event. If so, the kernel wakes up the waiting process, returning it the new event value that caused it to be woken, and then adds the wakeup increment to the event value.

A flag (bit 15 of the event function sub-code) is passed with each call to change the event value. This flag indicates to the kernel if it should wake up all processes waiting on the event for which the new value falls in the process's specified range (group wakeup), or only the first such process in the queue of processes waiting on the event (individual wakeup). For an individual wakeup the kernel wakes up the first process in the queue for which the event value is now in range - this may not be the first process in the queue. Processes are queued on an event in the chronological order in which they made the "wait for event" calls.

Note that the wakeup increment is added to the event value as soon as a process is woken, changing the event value for the check of subsequent processes in the queue during a group wakeup. Also note that processes earlier in the queue are *not* rechecked if the wakeup increment causes a change to the event value. It is therefore possible for a process to remain in the queue even though the event value now falls within its range.

The event value can be changed in four ways:

- a) Set a new absolute value.
- b) Specify a signed integer to add to the event value – "set relative".
- c) "Signal" the event – adds the "signal increment" to the event value.
- d) Pulse the event temporarily to an a new absolute value.

Each call to change the event value takes the individual/group wakeup flag (as described above). The calls are returned the event value as it was before the call, in the **d1.1** register⁶. If the "pulse" feature is used, the kernel sets the new event value and checks the queue of waiting processes in the normal way, but restores the event value back to its previous value before returning to the caller.

A process can wait for an event in two ways. Both specify a range – signed minimum and maximum values. One method specifies the range as absolute values. The other specifies the range as values relative to the current event value. The process will be woken when the event value is changed to fall within the specified range, unless the change is made with the "individual wakeup" flag, and the waiting process is not the first in-range process in the queue on the event. Also, due to the sequential nature of the group wakeup (described above), and to the immediate wakeup (described below), if the wakeup increment is not zero it is possible for a process to remain in the event queue even though the event value now falls within the desired range.

If a process attempts to wait for an event that is already within the specified range, the system call returns immediately to the caller. In this case the wakeup increment is applied to the event value as usual, but the kernel does *not* check the event queue to see if the new event value (assuming the wakeup increment is non-zero) is now in range for any waiting process.

⁶ From OS-9 version 2.3 onwards.

A single system call **F\$Event** is used for all the event functions. The required function is specified by a sub-code. The C library provides separate C functions for each event function. The following table shows the functions available, with the C function name and the assembly language sub-code name. The sub-codes are defined in the file 'DEFS/funcs.a'.

<u>Sub-code</u>	<u>C function</u>	<u>Description</u>
Ev\$Creat	_ev_create	Create an event (must not already exist).
Ev\$Delet	_ev_delete	Delete an event (link count must be zero).
Ev\$Link	_ev_link	Link to an existing event.
Ev\$UnLnk	_ev_unlink	Unlink from an event.
Ev\$Wait	_ev_wait	Wait for event – absolute maximum and minimum values.
Ev\$WaitR	_ev_waitr	Wait for event – maximum and minimum values relative to the current event value.
Ev\$Set	_ev_set	Set new event value.
Ev\$SetR	_ev_setr	Add signed quantity to event value.
Ev\$Signl	_ev_signal	Add "signal increment" (set when event created) to event value.
Ev\$Pulse	_ev_pulse	Momentarily change event value.
Ev\$Read	_ev_read	Read current event value.
Ev\$Info	_ev_info	Read (next) event table entry structure.

A process waiting on an event may be woken by a signal. This is not considered an error – the process is returned the event value at the time the signal is received, and can detect that it was woken by a signal (rather than by an appropriate event value) in one of two ways:

- a) The program's signal handler function sets a flag.
- b) The returned event value is not within the requested range (the program is returned the event value that existed when it made the "wait" call).

The "wait for event" function does *not* clear the process's signal mask (**P\$SigLvl** field of the process descriptor). If the signal mask is not zero, a pending signal will keep the process active, or a subsequent signal will wake the process, but the process's signal handler function will not be called until the process clears the signal mask.

Note that prior to OS-9 version 2.3 the "wait for event" function did not check whether a signal was already pending – the process would be put to sleep even if a signal was received while the "wait for event" function was

being executed. A process wishing to wait for an event *or* a signal to occur might not have responded to the signal until the event occurred.

8.6.1 Using Events

It may be seen from the above description that events are very flexible, and can be used in many different ways. Indeed, the only problem with using events is in deciding what technique is appropriate to a given situation. This section aims to reduce the possible confusion by describing some typical techniques.

The features available with OS-9 events have been very carefully chosen⁷. A very wide range of simple and complex inter-process synchronization algorithms can be implemented by a simple use of events, if the method of use is carefully chosen. In general, only a very few event statements are needed for even very complex algorithms. Therefore if your implementation of the algorithm in your application appears to require a complex or convoluted use of events it is worth reconsidering your approach.

In choosing a particular technique the principal aim must be secure operation (as with all forms of inter-process communication). That is, there must be no possible condition under which the mechanism will fail and cause the application to lock up or lose data. The features of OS-9 events are designed to give this security, provided they are used correctly. As with all system calls, event functions are indivisible – another process cannot be scheduled in while the call is executing, unless the kernel explicitly goes to sleep (such as during a "wait for event" call). Also, the kernel masks interrupts during critical code fragments, so events can be used from interrupt handlers (but not "wait for event"!).

8.6.2 Pulsing an event

In its simplest form an event can be used in the same way as a signal, except that an event does not cause the asynchronous execution of a handler function, nor can it wake a process that is not waiting on the event. One process creates the event, and another process links to the same event. The initial event value is set to zero, and the wakeup increment is set to zero. The process "receiving" the event waits for the event to reach a value of one (minimum value of range is one, maximum is one). The process "sending" the event pulses the event value to one.

⁷ Although one or two useful functions are not available, such as an indivisible "set event and wait".

In this simple technique the event is transitory – its value is always zero, except during the "pulse event" system call. In fact, this use of events is not secure – event changes cannot be made pending (equivalent to masking signals), so a process could decide to wait for an event that has already occurred. For this reason the "pulse event" technique should always be used with some form of handshaking, so that the "sending" process does not "send" the event until the "receiving" process is ready.

8.6.3 Interlocked handshake

A common use of events provides an interlocked handshake between two processes. For example, one process may place a data item in a data module, wake up another process by signalling an event, and then wait for the process to take the data. This can be done with complete security by using positive and negative event changes.

The event is created with a value of zero, and signal and wakeup increments of zero. The receiving process waits for the event to have a value of one (minimum is one, maximum is one). The sending process increments the event by one, changing the event value to one, which wakes up the receiving process. The sending process then waits for the event to have a value of zero.

When the receiving process has taken the data, it decrements the event value by one (adds minus one), changing the event value back to zero and waking up the sending process. There is no need for masking (which is not available with events) because the event value persists until changed, and a process attempting to wait on an event whose value is already in the desired range is immediately re-activated. Notice that this mechanism requires only two event statements in each program – a "wait for event" and a "set event relative". The equivalent algorithm implemented with signals would be significantly more complex.

8.6.4 Buffered handshake

The interlocked handshake described above is for the limited case of a single item of data being passed by the handshake. The event statements changing the event value could have explicitly set the event to one and zero instead of incrementing and decrementing it. It is in fact a subset of the more general case of a buffer of several items.

To enable a continuous flow of data, a buffer of two or more items may be used. As in the case of a single item, the buffer could conveniently be in a data module. For a multi-item buffer the sending process should only wait if

the buffer becomes full, and the receiving process should only wait if the buffer becomes empty. The mechanism is the same as for the single buffered case above. The event is created with a value of zero, and wakeup and signal increments of zero.

The sending process waits for the event to be in the range zero to "one less than the buffer size" – it will already be in this range unless the buffer is full. The sending process then adds the new item to the buffer and increments the event. The receiving process waits for the event to be in the range one to "the buffer size" – it will already be in this range unless the buffer is empty. The receiving process then takes the first item out of the buffer, and decrements the event.

Care must be taken in the use of variables for manipulating the buffer. An effective mechanism is to use a circular buffer in a data module, with next-in and next-out indices also in the data module. Only the sending process updates the next-in index, and only the receiving process updates the next-out index.

8.6.5 One to many synchronization

In this example one process writes data to a global pool (for example a data module), and multiple processes read the data. The sending process must not write new data until all receiving processes have read the data. The sending process wakes up all processes that were waiting for the data, and then itself waits until they have all accepted the data. The mechanism described below will work for any number of receiving processes, including zero (which can be useful for test purposes).

The event is created with a signal increment of minus one and a wakeup increment of one. The sending process writes new data to the data module, then sets the event to some large value (greater than the maximum number of waiting processes) – 1000 in this example. This wakes up all the waiting processes – they have been waiting for the event to have a value equal to or greater than 1000. Note that the call to change the event value specifies the **EV_ALLPROCS** group wakeup flag, to wake all waiting process for which the event value is now in range (a flag of zero would be specified for a single-process wakeup).

Because the wakeup increment is one, the event value is now equal to 1000 plus the number of processes that were waiting (and have now been woken). The sending process now subtracts the same large number (1000) from the event value. If the receiving processes have not yet changed the event value,

it will now be equal to the number of processes that were waiting. The sending process waits for the event value to be zero – which will already be true if no processes were waiting.

Each receiving process takes the data, and then decrements the event value – in this example by using the "signal increment" of minus one set when the event was created, for convenience. Once all the receiving processes have taken the data the event value is reduced to zero, and the sending process is woken. This final wakeup changes the event value to one, but this does not affect the mechanism, as the sending process will set it to an absolute value of 1000 when new data is ready.

This algorithm requires that the receiving processes be ready and waiting when new data is available. This is a common requirement where the sending process is gathering data at a fixed rate, and cannot delay if a receiving process is not ready. A check (such as a packet number in the data) would be used by the receiving processes to report an error if data is missed.

The sending process:

```
new_data();                /* write the new data */
/* Wake up all waiting processes: */
_ev_set(event_id,1000,EV_ALLPROCS); /* value = 1000 */
_ev_setr(event_id,-1000,0);
/* Value now = number-who-were-waiting */
_ev_wait(event_id,0,0);    /* wait until value = 0 */
/* Event value is now one (after our wakeup increment) */
```

The receiving processes:

```
/* Wait until the value is set: */
_ev_wait(event_id,1000,5000);
take_data();                /* get the new data */
_ev_signal(event_id,0);     /* decrement the value */
```

If the sending process must wait for all the receivers to be ready for the data, a modified form of the "interlocked handshake" described above can be used. The sender must know the number of receivers – stored in the variable **rx_num** in the example below. The event is created with a value of zero, a wakeup increment of zero, and a signal increment of one:

The sending process:

```
_ev_wait(event_id,rx_num,rx_num); /* wait for all receivers*/
new_data();                /* write the new data */
/* Wake up the receivers to take the data: */
_ev_set(event_id,0,EV_ALLPROCS); /* value = 0 */
```

The receiving processes:

```
_ev_signal(event_id,0);          /* increment the event */
_ev_wait(event_id,0,0);         /* wait for the data */
take_data();                    /* get the new data */
```

In effect, this is a many-to-one synchronization, with many receivers indicating their readiness to one sender, followed by a one-to-many synchronization, with the sender broadcasting its readiness to all the receivers.

8.6.6 Rendezvous

Two or more processes may need to know that they are at a common point in their programs, waiting until all processes are at this "rendezvous". In the following example the variable **procs** holds the number of processes that wish to rendezvous. The event is created with a value of zero, a signal increment of one, and a wakeup increment of zero. Each program uses the same code fragment:

```
_ev_signal(event_id,EV_ALLPROCS); /* increment the event */
_ev_wait(event_id,procs,procs);   /* wait for all processes */
```

After the rendezvous the event value must be reset back to zero by one of the processes:

```
_ev_setr(event_id,-procs,0);      /* reset the event */
```

There exists the possibility that this process, woken as a result of the event value change caused by the last process to come to the rendezvous, will reset the event value before that last process is able to execute its "wait for event" function. This problem exists because the OS-9 events system does not provide a single call to "change the event value and wait". This problem may be ameliorated by giving a low process priority to the process that resets the event value, so it is unlikely to "cut in" to the instruction sequence of the last process to join the rendezvous.

8.6.7 Semaphore

An event can be used to control access to a shared resource, such as a data module. A process wanting to use the resource must be made to wait until the resource is free. A process finishing with the resource must wake up the first process in the queue of processes waiting to use the resource. The event is effectively used as a "lock" or "semaphore" on the resource.

The event is created with a value of zero, a wakeup increment of one, and a signal increment of minus one. A process wanting to use the resource waits

for the event to have a value of zero. The wakeup increment automatically sets the event value to one, locking out any other process wanting to use the resource. When the process has finished with the resource it "signals" the event, setting it to one, using the "single process wakeup" mode so that only the first process in the queue is woken.

```
_ev_wait(event_id,0,0);          /* wait for the resource */
/* The event value is now one: */
use_resource();                 /* make use of the resource */
_ev_signal(event_id,0);         /* unlock the resource */
```

8.7 PIPES

A pipe is a "first-in-first-out" (FIFO) data store managed by the operating system. One or more processes write to the pipe using the standard I/O writing functions, and one or more processes read from the pipe using the standard I/O reading functions. The data is a byte stream – bytes are read from the pipe in the chronological order in which they were written to the pipe. Once one process has read a byte, the byte is lost from the pipe. When as many bytes have been read as were written, the pipe is empty – more bytes must be written before any more can be read. The pipe is of finite size. It becomes full if the number of bytes written exceeds the number of bytes read by the pipe size – no more bytes can be written until some have been read.

Because data is lost once read, and is read in strict chronological order, a pipe normally has only one reading process even if there are multiple writing processes. Multiple reading processes cannot know which process will read which data element, unless some other synchronization mechanism (such as an event) is used.

OS-9 pipes are memory buffers only (they are not held on disk). Pipes are managed by the **pipeman** file manager. Because pipes are held in memory there is no need for a device driver to manage an I/O interface. However, the OS-9 I/O system requires a device driver for every device. Therefore the device driver **null**⁸ is needed in memory for pipes to operate, but its functions do nothing. The device descriptor for pipes is **pipe**. Pipes are created by creating a path to the device '/pipe' using the normal I/O path creation functions.

A pipe may be created as a "named" pipe or as an "unnamed" pipe. An unnamed pipe is created if the path is created on the device name alone, by a "create" or "open" system call:

⁸ **pip**er prior to OS-9 version 2.3.

INTER-PROCESS COMMUNICATION

```
path=create("/pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);  
path=open("/pipe",S_IREAD|S_IWRITE);
```

A named pipe is created if the path is created with a second name element, using the "create" system call – similar to a single-level disk directory:

```
path=create("/pipe/fred",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);
```

Once a named pipe has been created it can be opened using the same path name:

```
path=open("/pipe/fred",S_IREAD|S_IWRITE);
```

By default the pipe file manager uses spare room in the path descriptor for the pipe buffer – 90 bytes. However, specifying an "initial file size" when creating the pipe (as would be done for a disk file) causes the file manager to allocate a separate buffer of the requested size, so pipes can be as large as needed. The actual size of the pipe buffer allocated may be greater than the requested size – the request is rounded up to the nearest multiple of the process minimum allocatable block size (16 bytes). Thus:

```
path=create("/pipe",S_IREAD|S_IWRITE|S_ISIZE,S_IREAD|S_IWRITE,1000);
```

will create a pipe buffer of 1008 bytes.

The pipe file manager connects multiple paths open on a named pipe so that they refer to the same memory buffer. In this way multiple processes can read and write the same pipe. The only way for multiple processes to access the same unnamed pipe is for the processes to inherit the path to the pipe, by being forked by a process that already has a path open to the pipe. That is, multiple paths cannot be open to an unnamed pipe – only multiple duplications of the same path permit multiple accesses to the pipe.

Implicit in this basic distinction between named and unnamed pipes are several differences in the details of operation, which are described below. Unnamed pipes are essentially a mechanism for connecting the standard output of one process to the standard input of another without the need for the processes to know that the path is not to a terminal. The features of pipes, and unnamed pipes in particular, reflect this requirement.

Pipes provide for data passing as well as inter-process synchronization. Reading from an empty pipe causes the process to be suspended until another process writes to the pipe, unless no other paths (or duplications of this path) have the pipe open for write, in which case the reader is returned an end-of-file error. This gives automatic synchronization between connected processes, with a proper end-of-file condition.

A process writing to a pipe when there is insufficient room in the pipe for the requested number of bytes is put to sleep until sufficient room becomes

available (because data has been read from the pipe by another process). To prevent a process "hanging up", writing to an unnamed pipe with no other paths (or duplications of this path) having it open for read returns a write error (**E_WRITE**). Thus if the connected reading process dies abnormally (for example, it is killed by the user), the writing process receives an indication of this condition, and can report an error or terminate itself.

This does not apply to named pipes. A named pipe can remain in existence even if there are no paths open to it, provided it contains data. Therefore a process attempting to write to a full named pipe is put to sleep even if no other process currently has the pipe open for reading, in the anticipation that another process will subsequently open a path to the pipe and read the data. A named pipe is automatically deleted if there are no paths open to it and it contains no data. It may also be deleted using the normal "delete" operating system call, provided no paths are open to it:

```
$ del /pipe/fred
```

The pipe file manager also permits the opening of a directory path on the '/pipe' device, allowing the single-level directory of named pipes to be read:

```
$ dir /pipe
```

If a process receives a "deadly" signal while waiting for a pipe operation to complete, **pipeman** will abort the operation, and return the signal code as an error code.

It may be seen that while pipes most resemble an **SCF** device (such as a terminal), named pipes have some of the properties of disk files. However, remember that the data has only a transient existence, and may only be read in the order in which it was written.

The "Get Status" call **SS_Ready** can be used to find out how many bytes of data are waiting in the pipe (just as for an **SCF** device). This call is made by the C function **_gs_rdy()**. Similarly, the "Set Status" call **SS_SSig** requests that the process be sent a signal when data is available in the pipe. This call is made by the C function **_ss_ssig()**.

8.7.1 Using Unnamed Pipes

As described above, because an unnamed pipe cannot be opened by name, only one path can be open on an unnamed pipe – the path created when creating the pipe. Therefore multiple processes can only access the same pipe by means of duplications of the path – that is, a process must inherit the path from its parent. If a process forks multiple children, or a child forks

INTER-PROCESS COMMUNICATION

another process (a "grandchild"), multiple processes can have access to the same pipe.

The **shell** uses unnamed pipes for piping the output of one process to the input of another. For example:

```
$ dir -ud ! grep -v "/"$ ! del -z
```

would be used to delete all files in the current data directory, but not attempting to delete sub-directories. The following sequence of operations forks two processes, with the standard output of the first process redirected to a pipe, and the standard input of the second process redirected to the same pipe. The original standard input and output paths of the parent are restored to their original paths. To simplify the example all error handling has been omitted – in practice every function call should always be checked for an error being returned.

```
copy_in=dup(0);          /* duplicate standard input path */
copy_out=dup(1);         /* duplicate standard output path */
close(1);                /* close standard output path */

/* Open the pipe. It is guaranteed to be path 1 (standard output), as
   the kernel uses the lowest available path number: */
path=create("/pipe",S_IREAD|S_IWRITE,S_IREAD|S_IWRITE);

/* Fork the first process, passing three paths: */
pid_1=os9exec(os9fork,"prog1",args1,envirom,0,0);

close(0);                /* close standard input path */
/* Duplicate the pipe. The duplicate is guaranteed to be path 0
   (standard input path): */
dup(1);

close(1);                /* close standard output path (the pipe) */
/* Duplicate the duplicate of the original standard output path,
   restoring the original standard output path: */
dup(copy_out);

/* Fork the second process, passing three paths: */
pid_2=os9exec(os9fork,"prog2",args2,envirom,0,0);

close(0);                /* close standard input path (the pipe) */
/* Duplicate the duplicate of the original standard input path,
   restoring the original standard input path: */
dup(copy_in);

/* Close the duplicates of the standard paths: */
close(copy_in);
close(copy_out);
```

8.7.2 Using Named Pipes

Named pipes allow a public use of pipes, and remove the requirement for the pipe to be an inherited path. They can also be used in applications where an unnamed pipe cannot be used – for example, a program may take a path name as an explicit parameter, rather than sending output to the standard output path. Within a multi-tasking application a named pipe can be opened only as needed. For example, an error logging process may take input from a named pipe, which it creates and keeps open. Other processes needing to report an error can open the named pipe, write to it, and then close the pipe. To prevent an "end of file" error when attempting to read from the pipe, the error logging process must duplicate the path (**dup()** C library function), so that the local path number used for reading is not the only incarnation of the only path open to the pipe.

Named pipes can help in debugging a multi-tasking application. The programmer can display a directory listing of all named pipes to see how much data is in each pipe (this appears as the "file size" in the directory listing):

```
$ dir /pipe -e
```

The programmer can also insert data into the pipe, simulating information being sent from another process:

```
$ echo "action 2" >+/pipe/commands
```

(note the use of the '>+' redirection to send data to an already existing pipe or file). The programmer can also read the contents of a pipe (but remember that the data is then lost to the application):

```
$ dump /pipe/info
```

Both named and unnamed pipes can be created with an explicit buffer size, overriding the default of 90 bytes. This should be done with care. It is reasonable to use a large buffer if the data structures being passed are large, but it is generally inadvisable to create a buffer that can hold a large number of data structures with the aim of relaxing the response time requirement on the reading process. The reason is that if the reading process cannot keep up with a small buffer, a large buffer will only allow large processing delays and will not prevent the eventual failure of the reading process to respond in time. However, it is reasonable to use a large buffer if the average rate of data is low, but the peak rate can be high – the pipe will absorb the peaks.

One of the potential problems in using pipes in a multi-tasking application is the reading process not responding in time, so the pipe fills up and a writing process is put to sleep. This may be the desired operation, giving

inter-process synchronization, but in many applications it would destroy the real time response of the writing process, which must return to its task of data collection. Of course, if the reading process does not respond in time this may be considered a fatal error, but the writing process must know of the error and be able to report it.

To do this the writing process can use the `_gs_rdy()` function to determine how much data is already in the pipe. Subtracting this from the pipe buffer size gives the free space in the pipe. If this is less than the required amount the process reports the error and does not write to the pipe. Be aware that this sequence is not indivisible – if there are multiple processes writing to the pipe, a process may decide from its check that there is sufficient space to write to the pipe, but before it writes its data another process could write to the pipe and fill it up. (See the chapter on Multi-tasking for techniques on making a sequence of instructions indivisible).

The example below shows a named pipe being created with a defined size. The program checks that space is available in the pipe before writing:

```
#include <modes.h>
#define P_MODE (S_IREAD|S_IWRITE|S_ISIZE)
#define P_PERM (S_IREAD|S_IWRITE)
#define P_SIZE 1000          /* size of pipe buffer */
char *pipe="/pipe/fred";    /* name for named pipe */
char message[80];           /* buffer for data to write to pipe */
main() {
    int path;
    path=create(pipe,P_MODE,P_PERM,P_SIZE); /* create pipe */
    while (1) {
        get_data(message); /* build message to send */
        if (_gs_rdy(path)>P_SIZE-strlen(message)) /* enough room? */
            _errmsg(0,"Pipe overflow\n"); /* no */
        else
            write(path,message,strlen(message)); /* write message */
    }
}
```

8.8 DISK FILES

Disk files provide a sophisticated form of inter-process communication. Large amounts of data can be passed, and inter-process synchronization is provided by record locking. The data is not transient⁹ – contrast pipes – and is not lost on power-down, even if it occurs while the file is still open (the file structure maintained by the RBF file manager is very robust). The disadvantages are the lower reliability, slower access speed, and generally

⁹ Except a volatile RAM disk.

higher power consumption when compared to semiconductor memory (although the effective power consumption per bit stored is low for high capacity disk drives).

The record locking provided by RBF ensures inter-process synchronization. Note that record locking is not effective for C "file" operations – **fread()**, **fwrite()**, **fprintf()**, and so on – because these library functions maintain private buffers unknown to RBF. The name "record locking" derives from its principal application of databases. A database file (usually) holds an array of data structures known as "records". The aim of record locking is to prevent a process reading or – worse still – writing back stale data. For example, if record locking were not supported, the following disastrous sequence of events could take place:

- 1) Process A reads a record from the file.
- 2) Process B reads the same record.
- 3) Process A modifies the record and writes it back to the file.
- 4) Process B modifies the record and writes it back to the file, cancelling the modifications made by process A.

Record locking works as follows. Consider two processes which have the same file open, and one, which has opened the file in update (read and write) mode, performs a read. The other process will be queued if it attempts to read some or all of the same data, until the first process rewrites the data, or reads or writes a different part of the file (that is, reads or writes another record). Note that it is not sufficient for the first process to seek to another point in the file – it must read or write (or close the file) for the second process to be woken. Because RBF records the start position and length of the data read on each path, this record locking works correctly for any size of data structure ("record").

A potential problem with record locking is "deadlock". For example:

- 1) Process A reads a record from file 1.
- 2) Process B reads a record from file 2.
- 3) Process A attempts to read the same record from file 2, and is put to sleep by RBF.
- 4) Process B attempts to read the record from file 1 that was read by process A, and is put to sleep by RBF.

This would result in both processes sleeping forever, waiting for the other to release a record. RBF checks for this condition, and would return a deadlock error (**E_DEADLK**) to process B at step 4.

RBF implements another form of locking: end-of-file lock. If a process has a file open for write, and its file pointer is at the end of the file (so that the process is likely to be extending the file), another process reading the file will be suspended at end-of-file, rather than being returned an end-of-file error. The idea is that the second process should be made to wait until the first process has written more data to the file, rather than be told that there is no more data in the file. This gives a very useful inter-process synchronization during sequential file writing and reading, similar to a pipe. The sleeping process is woken with an end-of-file error if the first process closes the file.

8.8.1 RAM Disks

Because the OS-9 I/O structure separates the logical file management functions from the hardware control functions (into the file manager and device driver respectively), it is possible to use a "disk" device driver that actually manages an area of memory rather than a disk drive. The device driver considers the memory as an array of equal sized blocks – each block is the size of a "sector", as specified in the device descriptor. When RBF requests that a series of sectors be read or written, the device driver simply copies between the buffer supplied by RBF and the corresponding memory blocks in the "disk".

A "memory disk" (or RAM disk) has two important benefits:

- a) Very fast access.
- b) Provides all the functions of disk files (such as record locking) without the need for a disk interface or a disk drive (relatively unreliable, large, and power hungry).

The disadvantage of a memory disk is the relatively high cost per bit. Memory disks are therefore useful for providing inter-process communications facilities, the temporary storage of frequently required data, and the storage (in non-volatile memory) of permanent data in diskless systems.

The RAM disk driver provided by Microware (**ram**) supports both volatile and non-volatile memory disks. Volatile memory is memory that loses its contents when the power is removed. In general the main memory of a

computer is volatile – usually a type of memory known as "dynamic" RAM, which is low cost but uses too much current to be powered from a backup battery when the main power is removed. Non-volatile memory does not lose its contents when the power is removed. This is usually a special area of battery-backed, low power consumption "static" RAM, or ROM (ROM can be used for a read-only "memory disk").

The device driver decides that a volatile disk is required if the "port address" (**M\$Port**) field in the device descriptor is less than 1024. In this case when the device driver is initialized it allocates memory from the main system memory, using the standard **F\$SRqMem** memory allocation system call. The amount of memory is calculated from the parameters in the device descriptor – the number of sectors per track and the number of sectors on track zero added together, and multiplied by the sector size (fixed at 256 bytes). The device driver then initializes the memory as if the **format** utility had been used, creating an initialized empty "disk". When the device is terminated, the termination routine of the device driver de-allocates the memory.

Usually several volatile RAM disk device descriptors are provided (in the 'CMDS' or 'CMDS/BOOTOBJS' directory). All have the module name **r0**, with various memory sizes specified. The file is given a name indicative of the memory size – for example, 'r0 256k' would be for a RAM disk 'r0' with a size of 256k bytes. The choice of RAM disk size usually depends on how much memory can reasonably be set aside for this purpose. Typically the desired device descriptor would be loaded in the 'startup' file.

If the "port address" is greater than or equal to 1024, the device driver assumes it is the address of an area of non-volatile memory, not known to the operating system's memory allocation functions. An area of memory is not known to the operating system if it is not in the memory lists in the **init** configuration module. The device driver initialization function does not initialize the memory in any way if the device descriptor is marked as format protected (bit 0 of the **PD_Cntl** field is set) – the **format** utility must be used (provided the memory is writable) with an appropriate alias device descriptor that is not format protected.

If the device descriptor is not format protected, and the disk validation field (**DD_Sync**) in sector zero is not correct (it must be \$4372757A), the device driver initializes the memory. So the first time the memory is used, the device driver initializes it, creating an "empty" disk. On subsequent uses the device driver does not initialize the memory (unless the disk validation field has become corrupted), so files are preserved.

INTER-PROCESS COMMUNICATION

A ROM disk is a useful way of providing fixed data to programs on a diskless system, where the programs normally expect to be reading disk files. For example, the 'termcap' file required by the **umacs** editor could be stored in a ROM disk. A ROM disk can be created with the following sequence of operations:

- a) Load and initialize a volatile RAM disk device descriptor of the desired ROM disk size:

```
$ load BOOTOBS/r0_64k
$ iniz /r0
```
- b) Copy the desired files to the RAM disk:

```
$ dsave -eb100 /r0
```
- c) Save the entire RAM disk to a disk file:

```
$ merge /r0@ -b100 >rom_disk
```
- d) Program PROMs from the disk file.
- e) Make a new device descriptor for the ROM disk, with the "port address" set to the address at which the PROMs are accessed. This can be done by using the **moded** utility on a copy of the RAM disk device descriptor file, changing the module name and port address.

Because I/O sub-systems are dynamically initialized and terminated under OS-9, a volatile memory disk must be explicitly initialized (**I\$Attach** system call) to remain in existence with no paths open to it:

```
$ load r0_256k
$ iniz /r0
```

This is typically done in the 'startup' file. The RAM disk can be terminated (**I\$Detach** system call) using the **deiniz** utility. Note that both attaching a device and changing directory to it increment the device use count. So to get rid of a RAM disk (return its memory to the free pool) it must be "detached" (**deiniz** utility) as many times as it was explicitly "attached" (**iniz** utility) *plus* as many times as **chx** and **chd** were used on it. Once the RAM disk has been terminated all the files in the RAM disk are lost. A non-volatile memory disk (battery-backed RAM, or ROM) does not need initializing or terminating.

The **ram** device driver provided by Microware performs no data integrity checks, other than a check of the validation word in sector zero. A battery-backed RAM disk can become corrupted due to power failure or program errors, particularly in systems that do not have the SSM for

inter-task memory protection. If the integrity of files in a battery-backed RAM disk is important, it may be advisable to protect against undetected corruption by modifying the **ram** device driver to write a CRC (Cyclical Redundancy Check) with each sector, and to check the CRC when a sector is read. The OS-9 module CRC system call (**F\$CRC**) can be used for this purpose, in which case an additional three bytes must be allocated for each "sector" of memory. To make this modification requires the source code to the **ram** device driver.

8.9 DATA MODULES

All multi-tasking operating systems must address the need for memory areas accessible by multiple processes, for use as data pools, buffers, and common information structures. OS-9 offers an elegant solution by making use of the OS-9 memory module concept. Normally a module must be present at startup (in ROM, for example), or be loaded using the **F\$Load** system call. However, in addition a module can be created in dynamically allocated memory, using the **F\$DatMod** system call. Prior to OS-9 version 2.3 only a module of type "data" could be created in this way. The chapter on the OS-9 System Calls gives a detailed description of the **F\$DatMod** system call, including how to create data modules in coloured memory, and how to create modules of other types.

The **F\$DatMod** system call, available through the **_mkdata_module()** and **make_module()** C library functions, should be considered as a way of allocating a named memory area of any desired size. The module has a header, body, and CRC just like other modules, but it is the body only that is of interest to the programmer – this is the "allocated memory". The kernel creates the module such that the body is equal in size to the size requested by the **F\$DatMod** system call (so the module in total is slightly larger), and clears the body to zeros, which can be a useful initialization aid. The CRC is initially correct, although it becomes invalid once data has been written to the module. This is of no importance unless the module is to be saved, and later loaded from disk or blown into ROM.

Because a data module is created from dynamically allocated system memory, the program cannot know the address of the memory at compile time – the address of the data module is returned by the **F\$DatMod** system call. Therefore the memory must be addressed register indirect (assembly language), or by a pointer (C language). The program will usually maintain two pointers, one giving the address of the module header (for later

unlinking), and the other pointing to the body of the module – the "allocated common memory".

The creation of a data module for shared memory is more elegant and more public than the alternative of one process passing the address of a memory area to other processes. In addition, the data module approach is compatible with the System Security Module inter-task memory protection. If the SSM is being used on a system, a process cannot access memory that it has not itself allocated, even if it has the address. However, if it creates or links to a module the kernel adds the module's address space to the process's memory map, so the process can access it. Thus the use of a data module for shared memory is upwardly compatible with systems using the SSM, while passing the address of a memory area is not. Note that the permissions (public, group, and user) specified when the module was created control the access to the memory. If a process has only read or execute permission for the module, the memory management unit will be configured to give a bus error if the process tries to write to the module (provided the MMU has the capability, as is the case for the 68851, 68030, and 68040).

The most convenient way to create and use a data module is to define a single C structure containing all the elements required to be in the shared memory. The size value used to create the data module is then simply the size of the structure (**sizeof** keyword), and the pointer to the module body is a pointer to that type of structure.

While the **F\$DatMod** system call returns both the address of the module header and the address of the module body, the C library function **_mkdata_module()** only returns the address of the module header. From this can be calculated the address of the module body, because the kernel initializes the "execution entry offset" of the extended module header with the offset to the module body. The example below shows such a calculation.

Once a process has created the data module, other processes can link to it, just as they would link to any other module in memory, using the **F\$Link** system call, or the **modlink()** C library function. This returns the address of the module header. As with the **_mkdata_module()** function, the address of the module body is calculated by adding the "execution entry offset" in the module header to the address of the module header.

Data modules can also be created and linked to by operating system components, such as device drivers. This can be used to provide shared memory between device drivers, or between a device driver and a process. If the data module is to be used by a device driver, it is a useful technique to

build the data module name from the device "port address". For example, if the port address is \$FC480000, the data module name could be 'dmFC480000'. This allows other incarnations of the device driver to be active at the same time, controlling other devices of the same type, without a conflict of names between the data modules.

When a process (or operating system component) has finished with the data module – for example, when the process or device driver is about to terminate – it should unlink from the data module, using the **F\$UnLink** system call or the **munlink()** C library function. The creation of the data module sets the module's link count to one, and each link to the module increments the link count. Each unlink decrements the link count, just as with any memory module. Once the link count reaches zero the module is removed from the module directory and its memory is returned to the free pool. Again, the use of data modules should simply be seen as named memory allocation, with the memory being returned to the free pool when no longer needed.

The kernel does not keep track of the modules a process or operating system component has created or linked to. Therefore if a process does not unlink from a data module – perhaps because it has been killed by the kernel or another process – the data module will remain in existence. This is not fatal, as the data module can be identified by name, and unlinked by the user or another process. Alternatively, if the application is restarted it can detect that the previous incarnation was abnormally terminated, because it is returned a "module already exists" error (**E_KWNMOD**) when it tries to create the module. The new incarnation can either exit with an error, or link to the already existing module, and clear the module body to zeros (as if it had just been created).

The same mechanism can be used for communication between loosely bound processes. As each process starts up, it attempts to create the data module. If it succeeds, it knows that it is the first process to use the module, and so must initialize the module. If it fails (with the error **E_KWNMOD**), it knows that it is not the first process – it then links to the module, and does not initialize it. If this approach is used, then some synchronization mechanism – such as an event – is needed to prevent subsequent processes using the module body before it has been completely initialized by the first process. This precaution is not needed if the initialization is performed in system state, for example by a device driver, because rescheduling will not take place while execution is in system state.

INTER-PROCESS COMMUNICATION

A data module is only shared memory – it does not provide any inter-process synchronization. Therefore, unless access is always in system state, or the use of fields within the data module has been carefully designed to need no interlocks between processes, some independent synchronization mechanism such as an event or signals must be used. This is because a process's time slice can end at any time, including while it is reading or writing to a data module (or any shared memory), and another process that uses the data module could become the current process.

It is possible to alter the behaviour of the kernel's process scheduler (see the chapter on "Multi-tasking"), but in general such an approach is less flexible, more difficult to debug, and more likely to cause problems for future adaptations of the software under development. Certain 68000 instructions are indivisible, and these can be used in many applications to avoid the need for a synchronization mechanism. Reading and writing of words and long words, and the "bit change" instructions, are useful examples. The C compiler generates these instructions, as can be seen by inspecting the assembly language output of the compiler, or the necessary small functions can be written in assembly language (see the chapter on Microware C and Assembly Language).

The following example shows the creation of a data module, whose body is to contain a declared structure type. If the module already exists, the program links to it instead. The address of the body of the module is calculated, and a character string is copied to one of the structure elements. Note that the macro **mkattrevs** that builds the attribute and revision word for creating the module is defined in the file 'module.h'. As usual, for clarity all error handling has been omitted, except for the check that the reason for being unable to create the module was because a module of that name already existed:

```
#include <module.h>      /* module header structure declarations */
/* The module will have read and write permission for all processes: */
#define PERMS (MP_OWNER_READ+MP_OWNER_WRITE+MP_GROUP_READ
              +MP_GROUP_WRITE+MP_WORLD_READ+MP_WORLD_WRITE)
#define ERROR -1

/* These functions return a pointer to a module header: */
mod_exec *modlink(),_mkdata_module();

/* This is the structure the data module will contain: */
typedef struct {
    int msg_len;
    char msg_str[100];
} data_struct;
```

```

char *mod_name="data_module",    /* the name of the data module */
    *message;                    /* the message string to write */

main(argc,argv)
int argc;
char **argv;
{
    mod_exec *mod_ptr;           /* pointer to module header */
    data_struct *data_ptr;       /* pointer to module body */

    /* Try to create the data module: */
    if ((mod_ptr=_mkdata_module(mod_name,sizeof(data_struct),
        mkattrevs(MA_REENT,1),PERMS))==(mod_exec *)ERROR) {
        /* Couldn't create the data module: */
        if (errno!=E_KWNMOD)
            exit(errno);         /* fatal error */
        /* Module already exists - link to it: */
        mod_ptr=modlink(mod_name,0);
    }
    /* Calculate the address of the module body using the
       address of the module header, and the offset in the header: */
    data_ptr=(data_struct *)((char *)mod_ptr +mod_ptr->mexec);
    /* Write the message to the structure in the data module: */
    strcpy(data_ptr->msg_str,message);
    /* And the length of the message: */
    data_ptr->msg_len=strlen(message);
}

```

8.10 SHARED EXTERNAL MEMORY

OS-9 only "knows" about memory areas specified to it in the memory search lists of the boot ROM and the **init** module. Therefore memory can be "hidden" from the operating system. Examples of memory which might usefully be excluded from the memory lists are:

- a) Battery-backed memory for configuration parameters.
- b) I/O memory, such as graphics display RAM.
- c) Inter-processor communications mailboxes and buffers.
- d) Fixed inter-process communication data space (not recommended).

When considering whether to "hide" an area of memory from the operating system you should first consider declaring it as coloured memory. If the memory area is given a priority of zero in the memory list, memory from that area can only be allocated by specific reference to its colour. This approach is more portable than creating a program that "knows" the absolute memory

address of a special memory area. However, the operating system will use the first few bytes of the memory to link it into the free memory lists, and this may be undesirable for certain types of special memory.

If the System Security Module is not used, there is nothing preventing a process directly addressing such memory (or any memory location). A process may also directly access the registers of an interface chip, such as a parallel port. This is perfectly acceptable provided you are sure it will not conflict with accesses from other processes, or a device driver. However, if the SSM is used these areas of "hidden" memory are not normally mapped in to a process's permitted memory map, and the process will generate a bus error if it attempts to read or write in that memory area. Such memory can normally only be accessed in system state, when the memory management unit's protections are suspended.

However, because this could be a serious restriction in certain applications, OS-9 allows a process to gain permission to access any memory area by using the **F\$Permit** system call. This system call adds a memory area to the memory map of a process. The process can request any combination of read, write, and execute permissions (although the 68851 and the MMUs in the 68030 and 68040 only support read and read-and-write, so a request for execute permission gives read permission). The complementary system call **F\$Protect** requests that the memory area be removed from the process's memory map.

Note that these system calls are actually installed by the SSM during its initialization, and are not part of the kernel. At coldstart the kernel installs handlers for these system calls that simply test the first byte of the memory area:

```
tst.b    (a2)
```

so this is the action taken if the SSM is not in use¹⁰. These system calls are only permitted from a process created by a super user (a member of group zero), or that has changed its user number to zero (using the **F\$User** system call) – only permitted if the program module was created by a super user. These system calls are described in detail in the chapter on the OS-9 System Calls.

Be aware that the normal data and program caching hardware facilities of the processor (if any) will still be operational during accesses to memory revealed by **F\$Permit**. Therefore accesses to I/O device registers are likely to cause problems if the processor has a data cache, as the processor may return

¹⁰ Under OS-9 version 2.2 the kernel does not install default handlers for these calls, so if the SSM is not in use these calls return an "unknown service request" error (**E\$UnkSvc**).

a value from the cache rather than re-reading the desired register, unless external address decoding circuitry inhibits caching during accesses to the I/O device registers. (The kernel disables the processor data cache during I/O system calls, so device driver accesses to I/O device registers are not cached).

There is no C library function to make the **F\$Permit** system call, so the assembly language for a C-callable function is given below. The example shows a C program calling the assembly language function, followed by the function itself. The **F\$Protect** system call (which is rarely needed) takes the same parameters, except that the **d1** register is not used.

```
map_in()
{
    /* Map in 64k of memory at address $FC840000, requesting
       read and write permission: */
    if (f_permit(0x10000,S_IREAD|S_WRITE,0xfc840000)==ERROR)
        exit(_errmsg(errno,"Can't access memory\n"));
}

#asm
*   f_permit(size,perms,address)
*   The F$Permit system call requires:
*       d0.l = size of memory area to map in
*       d1.w = access permissions
*       a2.l = start address of memory area to map in
f_permit:
        move.l   a2,-(a7)           save register
        movea.l  8(a7),a2           get start address parameter
        os9      F$Permit           map in memory
        bcc.s    f_permit10         ..success
        moveq     #0,d0
        move.w    d1,d0             copy error code
        move.l    d0,errno(a6)      save it
        moveq     #-1,d0            show error
        bra.s     f_permit20
f_permit10
        moveq     #0,d0             show no error
f_permit20
        movea.l   (a7)+,a2          retrieve register
        rts
#endasm
```

8.11 ALARMS

Alarms are not strictly an inter-process communication mechanism, as they do not provide a means by which one process can communicate with another. Rather, they allow the clock tick interrupt handler function to communicate with a process.

A process installs an alarm using the **F\$Alarm** system call. This requests that the kernel send a signal of a specified code to the process at a future time. Two types of alarm are available – single shot, and cyclic (periodic). The single shot alarm sends a signal after a specified number of ticks have elapsed (relative alarm), or at a specified date and time (absolute alarm), and then cancels (deletes) itself. The cyclic alarm sends signals repeatedly at the specified interval of ticks, until the process explicitly deletes the alarm, or the process dies. The kernel automatically deletes all outstanding alarms for a process when the process dies.

A single shot alarm allows a process to implement a timeout, for example when waiting for data to arrive on a serial port. A cyclic alarm is a useful means of getting a process to execute a sequence of instructions at strict intervals, independent of the time taken to execute the instructions (provided it does not exceed the alarm interval!). A cyclic alarm can also be used as a watchdog timer – the signal intercept routine of the process checks whether the main program body has set a flag in time, before the alarm signal was received.

A process can have any number of alarms installed at any one time. The **F\$Alarm** system call returns a unique ID (actually the address of the alarm "thread execution block"), which is used to identify the alarm when deleting it. An alarm can be deleted using the **F\$Alarm** system call, preventing any subsequent signals being sent by the alarm. Passing zero as the alarm ID when deleting alarms causes all alarms belonging to the process to be deleted (the kernel makes this call when a process dies). Only the creator of an alarm (same process ID) or a super user process (group zero) can delete an alarm.

The information about an alarm is held in a "thread execution block" allocated by the kernel when the alarm is created (see the section on the Process Descriptor in the chapter on the OS-9 Internal Structure). The thread block is linked into a linked list of thread blocks, rooted in the System Globals. The linked list is ordered by execution time – the first entry in the list will be executed first, and so on. Alarms are inserted in the list when they are created, and cyclic alarms are re-inserted in the list after every execution, ready for the next execution.

For an absolute alarm – set by date and time – the alarm date and time are stored in Julian format. Absolute alarms can therefore only be set to a resolution of one second. For a relative or cyclic alarm the alarm time given to the call is added to the current value of the **D_Ticks** field of the System Globals (ticks since system startup) before being stored in the thread block. Relative and cyclic alarms can therefore be specified to a resolution of one

tick. Note that if bit 31 of the time given to the call is set this indicates that the time value is given in 256ths of a second. The kernel clears bit 31, and converts the time to the nearest tick. This avoids the need for the programmer to know the tick period of the system. The minimum time that can be specified is one tick. This will cause a relative alarm to execute at the next tick, and a cyclic alarm to execute every tick.

Alarms are not directly acted on by the kernel's tick interrupt handler. Instead, the tick handler wakes up the System Process (see the chapter on Multi-tasking), and the System Process sends the alarm signals. The System Process has the highest possible process priority (65535), so it is sure to execute as soon as any currently executing interrupt service routines have finished, and any currently executing system call has finished or gone to sleep – that is, before any other program can continue execution in user state. This means that from a programming point of view the effect is exactly the same as the signals being sent from the tick interrupt handler, but because the tick interrupt handler does not have to handle the alarms it executes more quickly, and so allows other interrupts to be serviced with less latency.

The System Process, once activated, and having checked the timed sleep queue, checks every alarm in the queue for relative and cyclic alarms, comparing the alarm time in the thread block with the current value of **D_Ticks**. If the alarm time has been reached (or passed), the System Process executes the thread block function (sends the alarm signal), and removes the thread block from the queue. If the alarm is cyclic, the System Process adds the cyclic period to the alarm time in the thread block, and re-inserts it in the queue. Otherwise it de-allocates the thread block. Once all entries in the queue have been checked (stopping at the first entry that does not need execution, as the queue is in time order), the kernel calculates how many ticks must elapse before the first alarm still in the queue (if any) is to be executed. If this is less than the current value of **D_Elapse** (set by the check of the sleep queue), the System Process updates **D_Elapse** with the lesser value, so that it will wake up when necessary to execute the alarm.

Having checked the relative and cyclic queue, the System Process then checks the absolute queue, comparing each alarm date and time with the current date and time in the **D_Julian** and **D_Second** fields of the System Globals. If the alarm date and time have been reached (or exceeded) the System Process executes the thread block function, and deletes the thread block. Once all entries have been checked, the System Process checks the **D_Elapse** field, just as for the relative and cyclic alarms.

INTER-PROCESS COMMUNICATION

If the system date and time are changed (by the **F\$STime** system call), the kernel forces the System Process to be active, causing a check of the alarm queues. Therefore any absolute alarms that have expired as a result of the change are immediately executed. Note, however, that as the **F\$STime** system call does not update the **D_Ticks** field of the System Globals (number of ticks since system startup), relative alarms are not affected by the date and time change.

The **F\$Alarm** system call is used for all the alarm operations, with a function code specifying which operation is required. Separate C library functions are provided for each of the alarm operations. The table below shows the function codes with their symbolic names from the file 'DEFS/funcs.a', the corresponding C library functions, and a brief description of each operation.

<u>Code</u>	<u>Name</u>	<u>C function</u>	<u>Description</u>
0	A\$Delete	alm_delete	Delete an alarm, given the alarm ID (or zero to delete all alarms of a process).
1	A\$Set	alm_set	Create a relative single shot alarm.
2	A\$Cycle	alm_cycle	Create a cyclic alarm.
3	A\$AtDate	alm_atdate	Create an absolute alarm, given a date and time in Gregorian format (YYYYMMDD, 00HHMMSS).
4	A\$AtJul	alm_atjul	Create an absolute alarm, given a date and time in Julian format (date as days since 2nd January, year -4712, and time as seconds since midnight).

Alarms should be used with care. As described above, a process should if possible have only one sequence of instructions to execute, and so under normal circumstances it should not need the asynchronous change of flow of control provided by signals. A program that makes regular use of signals (rather than for exceptional conditions) is likely to be overly complex. Consider whether instead the program could be broken down into two or more separate processes using events, or using signals only as a synchronization mechanism.

8.11.1 System State Alarms

The System Process does not know implicitly what action to take when an alarm must be executed. Instead, it uses the register stack frame image that was built in the thread block by the **F\$Alarm** system call. If the system call is made from user state, the **F\$Alarm** handler routine builds an appropriate

register stack image for an **F\$Send** (send a signal) system call – **d0.w** is the process ID (of the calling process), **d1.w** is the requested signal code, and the program counter is set to the address of the **F\$Send** system call. This causes the **F\$Send** system call to be made as the execution of the alarm. However, any subroutine can be called by the System Process.

When the System Process has determined that an alarm must be executed, it switches its user group and user number to those of the process that created the alarm (in the **P\$User** field of the System Process process descriptor). It then sets the exception abort stack and return program counter (**P\$ExcpSP** and **P\$ExcpPC**) for a clean return to itself (because the execution is in system state, so without this provision an exception in the execution would cause a system crash). Finally, it takes the registers from the stack frame of the thread block (**d0** to **d7** and **a0** to **a3** – **a4** is the address of the System Process process descriptor, **a5** is the address of the stack frame in the thread block, and **a6** is the address of the System Globals), and calls the subroutine whose address is in the program counter field of the stack frame (**R\$pc**).

When the subroutine returns, if the carry flag is set the System Process puts the error code in the **d1.w** register into the **d1.w** register of the stack frame (clearing the high word of **d1.l**). Lastly, it puts the returned Condition Codes register (**ccr**) in the stack frame (**R\$ccr**), thus setting the carry flag in the stack frame if there was an error. This is a convenience for future uses of thread blocks, as alarm thread blocks are never returned to the caller. However, because the register stack frame is modified in this way, and also might be modified by the execution subroutine, when executing a cyclic alarm the System Process actually makes a copy of the stack frame in the thread block (on its stack) and uses that for the execution (**a5** points to it), so any changes to the stack frame used for the execution do not affect subsequent executions of the cyclic alarm.

The result is that alarms work differently when installed from system state, such as from a device driver or kernel customization module. Instead of sending a signal, the alarm execution uses a register stack frame given to the **F\$Alarm** system call, which is copied to the thread block. The caller can therefore specify all the data and address registers used when the alarm is executed (except **a4**, **a5**, and **a6**, which are pre-defined – see above), and the address of the subroutine to call (in the program counter field of the stack frame – **R\$pc**). This allows operating system components to "hook" subroutines into the clock tick interrupt service routine, providing watchdog, timeout, and polling functions independent of any calling process. Note that as with user state alarms the alarm is executed by the System Process, not

directly by the tick interrupt service routine, and interrupts are enabled when the alarm is executed.

For example, a single shot alarm can be used to turn off a floppy disk drive motor when the drive has not been used for a certain time, and a periodic alarm can be used to poll for input from a device that cannot generate interrupts. Because the caller specifies the processor register values to use when the installed routine is called, the routine can access the static storage of the caller (such as the device static storage used by a device driver), using the same symbolic names. In many ways an alarm routine installed from system state is very similar to an interrupt service routine – it is called asynchronously to the main body of (for example) the device driver, and can share static storage with the main body. However, because it is called from a process (the System Process), the alarm routine will not be called during the execution of a system call – it cannot break into the execution of the main body of a device driver, for example. The execution of the alarm routine will be deferred until the system call finishes or goes to sleep.

A system state alarm routine is called with the processor in supervisor state, and so has all the responsibilities of any system state routine. Although the System Process changes the group and user in its process descriptor to that of the creator of the alarm, the routine is still called as a subroutine of the System Process – the current process is the System Process. Therefore the routine must not sleep in any way (sleep, wait for event, make an I/O request that might sleep, and so on), because this would suspend the maintenance of the timed sleep queue and other alarms. However, other system calls that are forbidden in interrupt service routines can be used, because the System Process is scheduled in as the current process in the normal way, so there is no possibility of breaking into a system call being made by another process.

Similarly, the normal system state hardware exception recovery mechanism applies. If a bus error or other hardware exception occurs, control is transferred to the address given in the **P\$ExcpPC** field of the process descriptor, with the stack pointer given in the **P\$ExcpSP** field (see the chapter on Exception Handling). By default the System Process sets these before executing each alarm for a clean return to itself, ignoring any hardware exceptions.

When an alarm is created, using the **F\$Alarm** system call, the thread block that is allocated is linked in to the linked list of thread blocks allocated by the calling process. When the process dies, the kernel deletes all outstanding alarms for the process. This applies whether the call is made from user or system state. However, this is normally undesirable in system state, as an

alarm installed by a device driver in response to an initialization caused by a path being opened by a process must not be deleted simply because that process has died – other processes may now have paths open on the device. This difficulty may be avoided by temporarily substituting the address of the System Process's process descriptor for the current process before making the **F\$Alarm** system call. The thread block will then be allocated to the System Process. Also, the group and user for the alarm will be that of the System Process – 0.0. The following code fragment shows an example of this technique:

```
* Alarm time and date are in d3 and d4, function code is in d1
    move.l  D_Proc(a6),-(a7)    save current process descriptor
    move.l  D_SysProc,D_Proc(a6) make System Process current process
    lea     AlarmHandler(pc),a0 point at alarm subroutine
    suba.w  #R$Size,a7         make room on stack for stack frame
    move.l  a0,R$pc(a7)        set routine address
    movem.l d0-d7/a0-a3,(a7)   set other registers for call
    movea.l a7,a0              copy stack frame address for call
    os9     F$Alarm            make system call
    move.w  sr,d2              save carry flag
    adda.w  #R$Size,a7         ditch stack frame
    move.l  (a7)+,D_Proc(a6)    restore current process
    move.w  d2,sr              restore carry (error flag)

* Alarm ID is in d0.l, unless carry is set.
```

Because the System Process never dies, the kernel will not automatically delete alarms that have been installed in this way. This is similar to other resources installed in system state. For example, a device driver that uses an alarm must make sure that the alarm is deleted as part of its termination routine, otherwise the System Process could attempt to call an alarm routine that is no longer in memory.

The C library functions mentioned above assume that the calls are being made from user state, and are not suitable for use from system state. Therefore if you are writing system state code (such as a device driver) in C, you will need to write your own C-callable alarm functions in assembly language. The writing of C-callable functions in assembly language is described in the chapter on Microware C and Assembly Language.

CHAPTER 9

MULTI-TASKING



Multi-tasking is very important in real time applications, and is essential for multi-user systems. Although traditionally these uses have required very different operating systems, the multi-tasking features of OS-9 are suited to both without compromise or limitation. Microware have designed a simple and very elegant scheduling algorithm that is quick to execute, gives great flexibility, but is very easy to use. The default method of operation gives a prioritized automatic "round robin" scheduler, but a number of options are available to alter the behaviour of the scheduler. In the most extreme case, the scheduler can be made to operate in a purely hierarchical prioritized mode, such as is commonly found in simple real time kernels.

9.1 OS-9 PROCESS SCHEDULING

All of the process scheduling features of OS-9 are in the kernel module. The aim of the scheduler is to permit multiple processes (tasks) to be requesting processor time, and to divide up the processor time between them. To do this, OS-9 maintains a linked list of the processes that are requesting processor time, known as the "active queue". Each such process will eventually get some processor time, unless one of the pre-emptive features of the scheduler is in use. An important aspect of the OS-9 scheduler is the concept of the "current process". The current process is the process that is actually running now (except for the execution of interrupt service routines). The current process is not in the active queue. It is known because the System Globals field **D_Proc** points to its process descriptor. Therefore the active queue is the list of processes that want processor time, but are not currently receiving it.

MULTI-TASKING

Processes not in the active queue are not run by the scheduler. A process must be moved to the active queue to be requesting processor time. A process can only cease to be active by its own request, such as a "wait for event" or a "sleep" (which may be executed from within a system call, such as an I/O call), or if the process is terminated by the kernel in response to a "kill" signal, or a hardware exception, or a signal received by a process that has not installed a signal handler routine. A process is put in the active queue when it is first forked (unless it is forked for debugging), when it receives a signal, or when the condition it is waiting for (such as an event) occurs. This is the function of the **F\$AProc** system call.

Scheduling can use the automatic (time-slicing) scheduler, or the pre-emption mechanisms described below, or a mixture. The main work of the scheduler is carried out when a process is put in the active queue. The scheduler must decide at what position within the linked list to insert the new process. As described below, this is done in such a way that the next process to execute is always at the head of the queue. Therefore when the time comes to switch processes (a task switch), the decision of which process to make the current process is a very simple one. The kernel removes the first process in the queue from the linked list, and makes it the current process. This is the function of the **F\$NProc** system call.

A process switch (call to **F\$NProc**) is only carried out when the current process makes a system call that suspends it (such as a sleep request, perhaps from within a device driver), or the current process dies, or the processor descriptor of the current process is marked as "timed out" when the operating system is about to return to user state (to continue execution of the current process) after a system call or an interrupt. In this last case, the current process is inserted into the active queue exactly as if it had just become active, before the first process in the queue is removed from the queue to become the current process. It is therefore possible for the same process to become the current process again, for example if the process has a high priority. Note that if the active queue is empty (the current process is the only active process), the kernel does not waste time re-inserting the process in the active queue – it ignores the "time out".

The tick routine of the kernel is called on each clock tick interrupt. It decrements the **D_Slice** field of the System Globals, which contains the number of ticks remaining in the time slice of the current process. If this field is now zero, the time slice of the current process has finished. The kernel sets the "timed out" flag (bit 5 of **P\$State**) in the process descriptor of the current process. It also resets **D_Slice** to one, rather than zero, in case

the current process is the only active process – it is thus given another tick of processor time.

The actual process switch is not executed until the kernel is about to return to executing the current process in user state, and notices that the current process is timed out. At this time the kernel executes the **F\$AProc** system call to put the current process back in the active queue (as described above), and the **F\$NProc** system call to start execution of the next process. This feature ensures that system calls – which are executed in system state – are indivisible. That is, the current process will not be switched out while it is executing a system call, unless the system call explicitly puts the process to sleep.

The **F\$NProc** system call starts the processor running the next process. It removes the first process in the active queue from the linked list, and makes it the current process. It then resets the **D_Slice** field of the System Globals, using the value in **D_TSlice** (ticks per time slice). This initializes the count of the number of ticks in the time slice for the process. If the **F\$NProc** system call finds that there is no process to run, the kernel will execute the 68000 **stop** instruction, causing the processor to stop executing instructions until an interrupt occurs – the kernel then checks the active queue again (the interrupt handler may have woken a process by sending a signal, or changing the value of an event). If a flag is set in the first compatibility byte of the **init** module, the kernel will not execute the **stop** instruction, so it simply loops, checking the active queue until it is not empty (this is to satisfy some processor boards that cannot support the **stop** instruction).

Note that the number of ticks per time slice is typically two. This is because the system can only resolve time to an integral number of ticks. A process switch does not necessarily happen on a tick interrupt. It will also happen if the current process goes to sleep, perhaps to wait for an I/O operation to complete. If the number of ticks per time slice were one, a process could become the current process just before a tick interrupt, and so get very little time. With two ticks per time slice the process will get at least one full tick (unless it goes to sleep, or is pre-empted during its time slice).

9.2 THE SCHEDULER FEATURES

The OS-9 scheduler implements four types of scheduling. These are described briefly below, and then in greater detail in the following sections. Any combination of the scheduling mechanisms can be in use together.

- a) "Round robin" automatic scheduling. The processor time is divided into "time slices", and each process in the active queue is given a time slice in turn. A priority value assigned to each process causes high priority processes to receive time slices more frequently than low priority processes.
- b) "Minimum process priority" process suspension. Processes with a priority value less than a designated threshold receive no time slices, even if they are in the active queue. The process will receive time slices if the threshold is lowered, or the process's priority is raised (**F\$SPrior**), so that the process's priority is no longer below the threshold.
- c) Pre-emptive prioritized scheduling. This is the scheduling familiar to real time kernel users. The highest priority active process remains the current process until it ceases to be active, or another process with a higher priority becomes active, or its process priority is reduced so that it is no longer the highest priority process, or the priority of another active process is increased above the priority of the current process. Only processes whose priorities are greater than or equal to a threshold are treated in this way. Processes with a priority below the threshold continue to be scheduled in the "round robin" manner, but receive no time slices while any process is active with a priority greater than or equal to the threshold.
- d) Single process pre-emption. This is a mechanism that hands over scheduling to the application programmer. A process is specified as the pre-empting or "seizing" process. Only this process will be given processor time, until the specified process ID is changed, or the mechanism is suspended by specifying a process ID of zero. The scheduler will not give processor time to any other process, even if the pre-empting process goes to sleep, or dies.

9.3 ACTIVATING A PROCESS

As described above, a process is put in the active queue by the **F\$AProc** system call. This is a privileged system call (it can only be made from system state), and it is normally only used by other system calls within the kernel, such as the **F\$Send** system call (send a signal). Note that the process being put in the active queue may already be active (for example, the current process at the end of its time slice), and may even already be in the active

queue (for example, when a process's priority is changed by the **F\$SPrior** system call).

The kernel maintains a "current system active queue age" value in the **D_ActAge** field of the System Globals. This is a long word value, initially set to **\$7FFF0000**, and decremented at the start of each call to the **F\$AProc** routine. If it decrements below zero, it is reset to **\$7FFF0000** (and the process descriptors in the active queue are updated, as described below). The term "system age" is perhaps a little misleading, as this value decreases as the system gets older!

The **F\$AProc** routine, called to insert a process in the active queue, decrements the system age, and then calculates a "scheduling constant" (as described below) for the process. It is this scheduling constant that is used to determine the position of the process in the active queue. A process will be placed in the active queue ahead of a process with a lower scheduling constant. Note that a process that is being inserted into the active queue will be inserted *after* any processes with an equal scheduling constant. Once a process has been placed in the active queue its scheduling constant (written to the **P\$Sched** field of its process descriptor) is not changed, unless the threshold of one of the scheduling pre-emptive mechanisms is changed, or the priority of the process is changed, or the system age is decremented below zero. In all cases the active queue is kept ordered by scheduling constant.

Normally, the scheduling constant of a process being put in the active queue is calculated by adding the (decremented) system age to the process's priority. Because the system age never exceeds **\$7FFF0000**, and the process priority is a 16-bit word (and so cannot exceed **\$FFFF**), the scheduling constant cannot exceed **\$7FFFFFFF**. The process descriptor of the process is unlinked from any queue it may be in (the sleeping queue, the waiting queue, an event queue, or even the active queue), the new scheduling constant is written to the **P\$Sched** field, and the kernel searches the active queue to find the place to insert the process.

The kernel also checks whether the new process has a higher process priority than the current process. If so, it marks the current process as "timed out" (sets bit 5 of the **P\$State** field of its process descriptor). This causes the process now at the head of the active queue to become the current process when the currently executing system call or interrupt handler finishes. The aim is to allow a high priority process that has just woken up to immediately become the current process without waiting for the current process to finish

its time slice, unless a higher priority process is already active. This feature can be very important in real time applications.

If any of the three pre-emptive mechanisms is in use, the behaviour of **F\$AProc** described above is modified. The following sections describe the effect of the automatic scheduling mentioned above, and the behaviour of the scheduler if the pre-emptive mechanisms are used.

9.4 AUTOMATIC SCHEDULING

The aim of the automatic scheduler is to share the processor's time among multiple processes according to the following principles:

- a) Time slices are shared evenly, not given in a block to one process, followed by a block to another process.
- b) A process priority mechanism is available, whereby a process of a given priority will receive more time slices than one of a lower priority.
- c) All process of the same priority receive the same share of time slices, on a "round-robin" basis.
- d) The mechanism must execute quickly, so that scheduling does not consume a significant fraction of the processor's time.

The OS-9 scheduler satisfies the principles described above. In order to execute quickly it uses a simple algorithm (described above) that cannot easily be expressed mathematically. As required, a high priority process receives more time slices than a lower priority process. The relationship depends on the *absolute difference* between the priorities. Thus two processes with priorities of 100 and 105 share time slices in the same ratio as if they had priorities of 50 and 55, or 5 and 10.

The sharing of time slices can be calculated as follows. If the lowest priority active process has priority A, and other active processes have priorities which are B, C, D, and E respectively *greater than* A, then the proportion of time slices going to processes other than process A – for example, process D – is:

$$(D+1)/(2+B+1+C+1+D+1+E+1)$$

while the proportion of time slices going to process A is:

$$2/(2+B+1+C+1+D+1+E+1)$$

This has the interesting corollary that if the lowest priority process (or processes) has a priority only one less than that of the process with the next highest priority, it will get the same proportion of time as that process.

Another important effect of this algorithm is that quite a small difference in priority between processes will produce a large difference in processor time allocation. For example, if two processes are active, and one has a priority that is five higher than the other, the first process will get 5.5 times as much processor time as the second process. However, this is not usually of great importance, as a typical multi-tasking real time application will have a group of low priority processes, all with the same low priority, and a group of high priority processes, all with the same much higher priority.

9.5 AN EXAMPLE OF SCHEDULING

Below is shown an example of time slicing between three processes. It gives an empirical demonstration of how the processor time would be divided between three compute-bound processes¹¹ at different priorities.

The system age starts at 60. There are three processes, two of priority 10, and one of priority 8. All three are continuously active during the 10 time slices observed. The top row of the table shows the system age at each successive time slice. It is decremented by one each time as the current process is put back in the active queue. The three rows below show the scheduling constant for each of the processes at each time slice. The current process is marked with a '►'. The priority of the process is shown at the left of the row, and the total number of time slices for which the process was the current process is shown at the right of the row. Note that at each time slice only the scheduling constant of the process that was the current process in the previous process is recalculated – because the current process is put back in the active queue before the first process in the active queue becomes the new current process.

	System Age										
Priority	59	58	57	56	55	54	53	52	51	50	Slices
10	69 ►	69	67	67 ►	67	64	64 ►	64	61 ►	61	4
10	► 70	68	68 ►	68	65 ►	65	63 ►	63	63 ►	60	4
8	68	68 ►	68	64	64 ►	64	60 ►	60	60	60	2

¹¹ Processes that are continuously working, and do not ask to go to sleep.

Remember that the scheduling constant is calculated by adding the system age to the process's priority, and that if the scheduling constant of a process being put in the active queue is equal to that of a process already in the queue, the new process is put in the queue behind the process already in the queue. For the sake simplicity, the above example assumes that all three processes were initially put in the queue at the same system age (60). In fact, because the system age is decremented before a process is put in the queue, this would not happen in practice.

The example shows two important results. Firstly, the two processes of the same priority received the same number of time slices, while the lower priority process received less time slices. And secondly, the time slices were very evenly distributed between the processes. This illustrates that despite using a very simple algorithm, the OS-9 kernel achieves the aims of an automatic round robin scheduler.

9.6 SCHEDULING PRE-EMPTION MECHANISMS

OS-9 provides three mechanisms for the programmer to pre-empt the round robin scheduler. The mechanisms are all controlled by changing user-writable values in the System Globals. This is done using the **F\$SetSys** system call (made by the C library function **_setsys()**). This system call must be used to modify these controlling variables, even if the system is not using the SSM, (in which case the program could write to the variables directly). This is because the kernel takes action when these variables are changed, to ensure a correct change in the behaviour of the scheduler.

9.6.1 Minimum Priority

This facility allows a group of low priority processes to be suspended (given no processor time), and re-activated at a later time. This mechanism uses the System Globals field **D_MinPty**. If a process with a priority less than the value in this field is put in the active queue (by the **F\$AProc** system call), its scheduling constant is set to zero, rather than calculating the scheduling constant from the system age and the process's priority. (Note that the normal calculation is used if the process is in system state – to allow it to complete a system call). Because the process's scheduling constant is zero, it is put at the tail of the active queue, along with the other processes whose priorities are below the "minimum priority".

The **F\$NProc** system call checks the priority of the process it is about to make the current process. If the priority is less than the value in the

D_MinPty field, it marks the process's process descriptor as "timed out" (bit 5 set in the **P\$State** field. In addition, if the process is not in system state, the kernel calls the **F\$AProc** routine to re-insert the process in the active queue (which will set its scheduling constant to zero, and put it at the tail of the queue), and takes the next process from the head of the queue to be the current process. The process is marked as "timed out" so that if it is in system state (processing a system call), a task switch will occur as soon as the system call finishes. This allows the process to finish a system call (which must be permitted, otherwise system resources could be locked up), but not to execute any more of its program.

In this way, any process that was already in the active queue before the **D_MinPty** field was set above its priority is allowed to finish any system call it is executing, and then is re-inserted in the active queue, with a scheduling constant of zero. If the **F\$NProc** routine finds that the process at the head of the active queue has a scheduling constant of zero, it acts as if the active queue were empty, by suspending the processor's execution of instructions. It does not need to check the rest of the queue, as the queue is always kept sorted by scheduling constant, so any other processes in the queue must also have a scheduling constant of zero.

From OS-9 version 2.3 onwards, if the kernel finds on task switch that the current process is the only active process, but its priority is less than the value in **D_MinPty**, it re-inserts the process in the active queue, and calls the **F\$NProc** routine to activate the next process. As there is no other process in the active queue, this causes the current process to be suspended (its priority is less than **D_MinPty**), and processor execution to be suspended. This guarantees that processes with a priority below **D_MinPty** are immediately suspended (after completing any system call). This may be needed to prevent these processes making a system call that takes some time to execute, possibly impairing the real time response of the high priority processes. Prior to OS-9 version 2.3, if the current process was the only active process it continued execution, even if its priority became less than **D_MinPty**.

The result of this algorithm, in conjunction with the fact that the current process is marked as "timed out" if a higher priority process is put in the active queue, is that a high priority process can set the **D_MinPty** field to immediately suspend a group of low priority processes, and then allow them to run at a later time by clearing the **D_MinPty** field.

To ensure that the low priority processes are re-activated when the **D_MinPty** threshold is lowered, the **F\$SetSys** system call (used to change

fields in the System Globals) takes special action if this field is being changed, and the new value is less than the present value. It scans through the active queue, and re-inserts any process whose current scheduling constant is zero (using the **F\$AProc** routine), causing its scheduling constant to be recalculated. It is therefore essential that the **D_MinPty** field is changed using the **F\$SetSys** system call or the **setsys()** C library function, rather than by directly writing to the System Globals, as otherwise the low priority processes will never be re-activated. Note that a process is simply re-inserted in the active queue when the "minimum priority" is lowered - it is not necessarily re-activated, because its priority may still be below the "minimum priority". This permits any number of groups of processes at different priority levels to be suspended and re-activated in a hierarchy.

9.6.2 Maximum Age

The term "maximum age" used to refer to this mechanism is something of a misnomer, as the mechanism acts on the process's priority, not its "age". (See the chapter on the OS-9 Internal Structure for a discussion of a process's "age", which is a value invented only when a copy of the process descriptor is requested.)

The "maximum age" field in the System Globals - **D_MaxAge** - sets a threshold. A process with a priority less than this threshold is scheduled in the normal "round robin" way, while processes with priorities greater than or equal to the threshold are scheduled in a strictly prioritized manner. If the **D_MaxAge** field is zero (the default on startup), this mechanism is disabled.

If **D_MaxAge** is not zero, and a process has a priority greater than or equal to the threshold, then the **F\$AProc** routine calculates its scheduling constant in a different way. Instead of adding the process priority to the current system age, it adds the process priority to \$80000000. As described above, the normal method of calculating the scheduling constant cannot produce a result greater than \$7FFFFFFF. Therefore all processes in the group with priorities equal to or above the threshold will always have scheduling constants greater than all processes in the lower group, and so any process in the upper group will be inserted in the active queue ahead of all processes in the lower group.

The first effect of this mechanism is that processes in the lower group will not run so long as any process in the higher group is active. The second effect is that the processes in the upper group that are active are always ordered strictly by priority in the active queue, irrespective of how much processor

time they have already used. This means that the highest priority active process will always be the current process. It must cease to be active (or have its priority changed) in order for the process with the next highest priority to become the current process. Therefore processes in the upper group are subject to a pre-emptive prioritized scheduling mechanism - there is no "round robin" distribution of processor time. This is the scheduler familiar to users of real time kernels.

Changing an active process's priority (using the **F\$SPrior** system call, or **setpr()** C library function) causes it to be re-inserted in the active queue, and if a process with a higher priority than the current process is inserted in the active queue, then the current process is marked as "timed out". Therefore, as with the "minimum priority" mechanism, processes in the upper group immediately pre-empt processes in the lower group. That is, if the current process is a process in the lower group, and a process in the upper group becomes active, the time slice of the current process is immediately terminated.

The **F\$SetSys** system call checks whether the **D_MaxAge** field is being changed. If so, it calls the **F\$AProc** routine to re-insert every active process back into the active queue. This ensures that a change in threshold is immediately acted upon, with a re-ordering of the upper and lower groups. Also, if the current process is now in the lower group, and any process in the upper group is active, the current process is marked as "timed out", as described above. It is therefore essential that **D_MaxAge** is changed by using the **F\$SetSys** system call or the **setsys()** C library function, rather than by writing directly to the System Globals.

9.6.3 Seizing Control

This mechanism completely pre-empts the scheduler, leaving all scheduling to be done by the application. It uses the System Globals field **D_Sieze** (note the spelling). The mechanism is enabled if this field is not zero, and is disabled again if the field is set to zero. When the mechanism is enabled, the **D_Sieze** field is assumed to contain the ID of a process. When the process with this ID is put in the active queue by the **F\$AProc** system call it is given a scheduling constant of \$FFFFFFFF, forcing it to the front of the queue. As described above, if it also has a higher priority than the current process, the current process is marked as "timed out".

When the current process is switched out, and the kernel looks for the next process to run, the **F\$NProc** system call will make the "seizing" process the current process (because it is at the head of the queue). At the end of its time

slice the process will again become the current process, because the **F\$AProc** routine will again force it to the front of the active queue. In addition, if the process goes to sleep (or even if it dies!), the **F\$NProc** routine will refuse to run any other process, and will suspend execution just as if the active queue were empty. This mechanism therefore leaves the scheduling entirely in the hands of the programmer, and clearly it must be used with extreme care. Indeed, because of the dangers involved, this mechanism should only be used if there is absolutely no alternative, which is extremely rare.

The **F\$SetSys** system call takes no special action when the **D_Sieze** variable is changed.

9.6.4 The Precedence of the Mechanisms

Although any or all of the mechanisms described above can be activated at any one time, in some respects they are clearly in conflict. It is therefore useful to know in what order of precedence the kernel acts on them.

The "seizing" mechanism has the highest precedence. If **D_Sieze** is not zero, the other mechanisms are inoperative. Otherwise, the priority of a process is first checked against the **D_MinPty** field, and only if it is not below this threshold, or the process is in system state, is the priority also checked against the **D_MaxAge** field. Therefore, if a process's priority is below **D_MinPty** the process will be suspended, even if its priority is equal to or greater than **D_MaxAge**, unless it is in system state (presumably executing a system call).

9.7 SCHEDULING IN REAL TIME APPLICATIONS

The processes in a typical real time application will be divided into two groups:

- a) High priority processes that are reacting to I/O events. These processes sleep, waiting for an I/O event, wake up to deal with the I/O event, and then go to sleep again. These processes are real time – they must respond to the I/O event within the specified time, or the system has failed.
- b) Low priority processes that are handling non-real-time functions. User interface and reporting processes usually fall into this category.

Although the pre-emption mechanisms described above are available, they are very rarely needed. In almost every case it is sufficient to give the first group of processes all the same priority, which is significantly higher than that of the second group, which also all have the same priority.

If one of the high priority processes is woken it will get processor time ahead of all the low priority processes, although it may execute after one or more other high priority processes. In addition, if a low priority process is the current process when a high priority process wakes up, the time slice of the low priority process is immediately terminated, so the high priority process immediately becomes the current process.

To make one process execute to the exclusion of all others for a short time it is only necessary to give it a significantly higher priority. For example, if the high priority group of processes has a priority of 1000, while the low priority group has a priority of 100, then a high priority process will (to a very rough approximation), get 900 time slices before any low priority process receives any processor time. As this typically equates to 18 seconds of processor time, the high priority process will have plenty of time to finish its job and go back to sleep, without worrying that it may lose processor time to a low priority process.

This mechanism is made even more flexible by the ability of a process to change its own priority, using the **F\$SPrior** system call (made by the **setpr()** C library function). In addition, a process can change the priority of another process, provided the process making the **F\$SPrior** system call is owned by the same user (same group number and user ID), or it is owned by a super user (group zero).

If a greater degree of control is required in very time critical applications, the "maximum age" pre-emption mechanism can be used. This retains the benefits of the automatic scheduling for the lower priority group of processes, while giving a deterministic prioritized scheduling for the upper group of processes.

Note that a task switch is not performed if the current process is executing in system state. This causes system calls to be indivisible, but it also means that task switching is suspended while a system state process is the current process. Because a system call is allowed to proceed to completion (or until it explicitly goes to sleep), a lengthy system call that does not sleep – such as a large disk transfer without DMA – can cause a significant delay before even a high priority process gets processor time. This should be taken into account when writing operating system components such as device drivers. The

MULTI-TASKING

device driver, knowing that it is taking a long time to complete its operation, could sleep for one tick (which causes the process to be immediately re-inserted in the active queue) at regular intervals, allowing other processes an opportunity to gain processor time.

The same caution should be applied to interrupt service routines. The execution of processes is naturally suspended while an interrupt is being serviced, because the interrupt causes the processor to change the flow of control. Therefore interrupt routines should be as short as possible, to avoid compromising the real time response of high priority processes.

CHAPTER 10

EXCEPTION HANDLING



68000 family exceptions are a class of processor operations that change the flow of control of the processor without losing the current state of the program. Each exception condition has a number from 0 to 255, identifying the particular exception.

The exceptions fall into three groups:

- a) Explicit program instructions – trap #n, TRAPV, CHK, CHK2, TRAPcc, and cpTRAPcc.
- b) Special events occurring during the execution of an instruction – Bus Error, Address Error, Illegal Instruction, Zero Divide, Privilege Violation, Trace, Line 1010 and Line 1111 Emulator, Coprocessor Protocol Violation, Format Error, and Coprocessor Exceptions.
- c) External signals – Reset, Auto-vectored Interrupts, and Normal Vectored Interrupts.

Groups (a) and (b) together are sometimes known as the "hardware exceptions". All exceptions cause the same sequence of operations:

- 1) The current program counter and status register are saved on the supervisor stack. The 68010/020/030/040 also save a stack format word and a vector offset word. Depending on the exception, other information may also be stacked. For example, internal state information is stacked on bus error, to allow virtual memory support (not the 68000). This operation is omitted for the Reset exception.

- 2) The status register is updated – the supervisor state bit is set, the trace bit is cleared, and the interrupt mask is set to the appropriate level (Reset and Interrupt exceptions only). The Reset exception also sets the Vector Base Register to zero (not the 68000).
- 3) The appropriate vector (exception handler routine address) is obtained from the exception vector table (indexed by the exception number), and put in the Program Counter. The Reset exception also sets the supervisor stack pointer from the long word at absolute location zero. For normal vectored interrupts the vector number (64 to 255) is read from the interrupting device. For other exceptions the vector number is generated internally (or provided by the coprocessor, for coprocessor exceptions).

10.1 EXCEPTION HANDLING UNDER OS-9

OS-9 provides default handling for all exceptions. It also provides mechanisms for programs or operating system components to handle any or all exceptions.

OS-9 creates four groups of exceptions:

- a) The **trap #0** instruction, used to make operating system calls.
- b) The other **trap #n** instructions (1 to 15), used to call trap handler modules.
- c) Exceptions as the result of instruction execution – the "hardware" exceptions, other than the **trap #n** exceptions.
- d) Interrupts.

OS-9 provides separate mechanisms for handling each of these groups. There are also two mechanisms that apply to all exceptions:

- a) Overwriting the exception jump table (always in RAM).
- b) Overwriting the exception vector table (may be in ROM, and therefore not writable).

These two mechanisms allow slightly more rapid access to exception handling routines, in particular interrupt service routines. However, because

the exception handling is not through the kernel, operating system calls must be used with extreme care. The **D_IOGlob** area of the System Globals can be used for the storage of variables – for example, it can hold a copy of the old vector, so the interrupt service routine can continue on into the kernel's interrupt handler if desired.

Typical uses are pseudo-DMA and software dynamic RAM refresh.

10.2 USER AND SYSTEM STATE RETURN

The "user state return" routine is called by the kernel after handling any exception if the processor was in user state before the exception, or a task switch has been performed and the current process is about to be executed in user state. The kernel performs the following sequence of operations:

- 1) The kernel tests the "timed out" flag in the process descriptor (and clears it, ready for the next time slice). If it is set, the kernel performs a task switch, unless the active queue is empty, in which case it allows the current process to continue execution. From OS-9 version 2.3 onwards, it first checks that the priority of the process is not below the "minimum priority" threshold (**D_MinPty** in the System Globals). Otherwise it inserts the process in the active queue, to force it to be suspended.
- 2) If the process was not timed out, the kernel checks whether the process is "condemned" (bit 1 of the **P\$State** field of the process descriptor is set) – the process has received a "kill" signal, or a debugged process has died. If so, it terminates the process, and calls the **F\$NProc** routine to make the next process in the active queue the current process.
- 3) The kernel checks whether the process has a signal pending (the **P\$Signal** field of the process descriptor is not zero), and the signal mask (**P\$SigLvl**) is clear. If so, the kernel terminates the process if it has no signal handler routine installed, otherwise the kernel copies the process's register stack frame to the process's user stack (so that execution continues with the main body of the program when the signal handler finishes, by making an **F\$RTE** system call), sets the signal mask to one, and modifies the register stack frame as follows:
d0 = number of signals in the queue (including the current

one).

d1 = the signal code.

d2 = zero.

a6 = signal handler's static storage.

pc = address of signal handler routine.

This causes the signal handler to be executed when the process is restarted.

- 4) If the system has a floating point unit, the kernel saves the current FPU context, and restores the FPU context of the new current process (unless there was no change of current process).
- 5) If the system is using the SSM, the kernel calls the **F\$AltTsk** routine, to ensure the MMU is correctly set up for the process.
- 6) The kernel clears bit 7 of the **P\$State** field of the process descriptor, to indicate that the process is executing in user state.
- 7) Lastly, the kernel restores the data and address registers from the register stack frame, adds 4 to the stack pointer to ditch the vector offset value (see the section on the Exception Jump Table), and executes the **rte** instruction. This instruction loads the status register and program counter from the stack, causing execution to continue with the instruction following the system call, or the point at which an interrupt occurred or the process was suspended by a task switch.

If the process is returning to system state (for example, an operating system component makes a system call, or a system call routine is interrupted), only step 7 is executed. It is this that causes system calls to be indivisible.

Additional system calls can be installed, or existing ones replaced using the **F\$SSvc** privileged system call. This is normally done by a kernel customization module, but may be done by any operating system component, such as a device driver or file manager.

10.3 SYSTEM CALLS - TRAP #0

This processor instruction is reserved in OS-9 for making operating system calls. A system call instruction has the form:

```

trap      #0
dc.w      function_code

```

The Microware assembler provides a built-in macro **os9** to do this in one instruction. For example:

```

trap      #0
dc.w      F$Link

```

can be expressed as:

```

os9       F$Link

```

When this exception occurs, the kernel reads the function code word pointed to by the saved program counter (on the supervisor stack), and then adds two to the saved program counter, updating it to point at the instruction following the function code.

The kernel uses the function code word as an index into either the User or the System Dispatch table, depending on whether the call was made from user or supervisor state respectively (the kernel tests the supervisor state flag - bit 13 - of the saved status register on the supervisor stack). The table entry is the address of the routine to call to handle the system call.

Unless the call is made from within an interrupt service routine, the stack used for the system call is naturally the System State stack of the calling process (the upper half of the Process Descriptor), because the processor automatically switches from using the user stack to using the supervisor stack when an exception occurs (because the supervisor state bit is set in the status register). When a process is forked, the kernel sets its system state stack pointer (in the **P\$sp** field of the process descriptor) to the address of the top of the process descriptor. When a process becomes the current process, the kernel sets the processor's supervisor stack pointer register from the **P\$sp** field of the process descriptor. When a process ceases to be the current process (it goes to sleep, or a task switch occurs), the kernel saves the processor's supervisor stack pointer register in the **P\$sp** field of the process descriptor.

The kernel's handler for the **trap #0** exception saves all the data and address registers, making a register stack frame on the process's system state stack. The stack frame includes not only the data and address registers, but also above them the vector offset (the vector number times four) as a long word in the **R\$a7** field, the status register (a word), and the program counter (updated past the **trap #0** instruction as part of the processor's exception handling). The kernel clears the condition codes register (the low byte of the status register) in the stack frame, so that the default is to return the carry flag clear to the caller, indicating no error.

EXCEPTION HANDLING

If the call is made from user state, the kernel also saves the supervisor stack pointer (pointing to the stack frame) and user stack pointer registers in the **P\$sp** and **P\$usp** fields of the process descriptor respectively. The kernel then saves the values currently in the **P\$ExcpPC** and **P\$ExcpSP** fields of the process descriptor, and sets them equal to the supervisor stack pointer and the address of the instruction following the kernel's call to the system call handler routine. This causes any hardware exception within the system call handler to return cleanly to the kernel, as described below.

The kernel copies the function code to the high word of the "stack pointer" field (**R\$a7**) of the stack frame (the low word contains the vector offset, placed on the stack as a long word by the exception jump table instructions), and sets the **a5** register to point to the stack frame. The system call handler routine will use the stack frame to access the caller's parameters, and to return values to the caller.

Finally, the kernel uses the function code, as described above, to get the address of the appropriate handler routine, and also to get the address of the static storage of the handler routine (see the chapter on the OS-9 Internal Structure). The kernel then calls the handler routine. On return from the handler routine (which could be as the result of a hardware exception), the kernel checks the returned carry flag. If it is set, the kernel sets the carry flag in the condition codes register in the stack frame, clears the high word of the **d1** register in the stack frame, and writes the error code (returned in the low word of the **d1** register) to the low word of the **d1** register in the stack frame. Note that this means that if there is no error, the **d1** register is preserved (unless it was changed in the stack frame by the system call handler routine), otherwise the **d1** register contains the error code as a long word (the high word is zero).

On return from the system call handler, and having set the error status in the stack frame if there was an error, the kernel branches to its "user state return" or its "system state return", depending on whether the call was made from user or system state.

10.4 TRAP HANDLER MODULES - TRAPS #1 TO #15

These processor instructions are used within OS-9 to call trap handler modules. A trap handler module is essentially an OS-9 memory module containing any number of subroutines that can be called by function code rather than by address, and that has its own static storage, separate from that of the process using the trap handler. This provides a mechanism for

calling functions without the need to know the address of the functions, or the need to reserve static storage for them.

Microware provide the **cio** and **math** trap handler modules. The C libraries, in conjunction with 'cstart.a', use **trap #13** and **trap #15** respectively to call these trap handler modules, although in principle any trap number can be used to call any trap handler module. The program wishing to use a trap handler module makes the **F\$TLink** system call, specifying the name of the trap handler module, the number of the trap instruction (1 to 15) that will be used to call it, and an "additional static storage" size (usually zero).

The kernel allocates static storage memory for the trap handler. The size of the static storage is the sum of the **M\$Mem** and **M\$Stack** fields of the trap handler's module header, and the "additional static storage" parameter to the **F\$TLink** system call. The trap handler may not require any static storage, in which case its **M\$Mem** and **M\$Stack** fields are zero. The kernel saves the addresses of the trap handler module and its static storage, and the size of the static storage, in the process descriptor of the calling process. The process descriptor fields used (**P\$Traps**, **P\$TrpMem**, and **P\$TrpSiz**) are each arrays of 15 entries, so a process can use up to 15 trap handlers concurrently, one for each **trap #n** instruction other than **trap #0**.

Note that the static storage for the trap handler is allocated separately for each process that has used the **F\$TLink** system call to link to the trap handler. The kernel initializes the static storage in the same way that it does for a program, so initialized data can be used. Also, the kernel adds 32k to the static storage address before saving it in the process descriptor, just as is done for a program. The linker compensates by subtracting 32k from all static storage references when creating a trap handler module. As for a program, this is done to maximize the amount of static storage that can be accessed using the signed 16-bit constant offset indexed addressing mode of the 68000 processor family.

A trap handler can execute in user or supervisor state. The kernel will execute trap handler functions in supervisor state if the "system state" bit is set in the attributes field of the module header. However, the kernel will give a "no permission" error (**E_PERMIT**) if an **F\$TLink** system call is made for a system state trap handler which was not created by a super user – that is, the owner group number (the high word of **M\$Owner**) of the module header is not zero.

Once a process has linked to a trap handler using the **F\$TLink** system call, it can call the functions of the trap handler using the **trap #n** instruction

EXCEPTION HANDLING

followed by a 16-bit function code. For example, to call the **sscanf()** function in the **cio** trap handler:

```
trap      #13
dc.w      $1A
```

Unless the trap handler is a system state trap handler, the kernel builds a parameter frame on the user state stack (as show below), restores the data and address registers, and uses the **rte** instruction to return to the state (system or user) of the caller and jump to the trap handler.

Therefore if a system state process calls a "user state" trap handler, the trap handler is called in system state. However, because the stack frame is built on the user state stack, the trap handler will have no access to the stack frame (unless it knows it is being called from system state, which it cannot check if it could also be called from user state, as reading the high byte of the status register in user state is only possible on the 68000/010). This implies that a system state process cannot successfully call a user state trap handler.

If the trap handler is a system state trap handler, the kernel builds a stack frame as described below, and calls the trap handler entry point as a subroutine (in system state). On return, the kernel calls its "return to user state" function. This implies that a system state process should not call a system state trap handler. Note that prior to OS-9 version 2.3, the kernel jumped directly to the trap handler entry point, so it was the responsibility of the trap handler to finish with an **rte** instruction. Since OS-9 version 2.3 the trap handler returns to the kernel with an **rts** instruction, allowing the kernel to perform its normal "return to user state" checks.

10.4.1 The Trap Handler Routine

Because a user state trap handler returns directly to the calling process, not through the kernel, it is the trap handler's responsibility to preserve or modify the processor registers as required. In effect, the trap handler is acting as a subroutine of the program. The kernel calls a user-state trap handler with the following register parameters and stack frame:

```
d0-d7/a0-a5 = caller's registers
(a6) = trap handler's static storage
a7 = usp
8(a7).l = caller's return program counter
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

The exception vector offset is the offset for the trap instruction vector, which is $(32 + \text{trap_number}) * 4$. If the trap handler has no private static storage, the

a6 register passed is the value used by the program when making the **F\$TLink** system call – usually the address of its own static storage. The trap handler must use the function code to decide which subroutine to execute. On return from the subroutine it must restore the caller's **a6** register from the stack frame, add 8 to the stack to remove the parameters, and execute an **rts** instruction to return to the program.

Because the trap handler is effectively acting as a subroutine of the program, it can make any system calls, which will be made on behalf of the program. However, C library functions must be used with care, as they may use private static storage variables. These static storage variables will be in the trap handler's static storage, not the program's, which might cause some conflict (for example, when using buffered I/O functions such as **fread()**).

A system state trap handler is called with registers and stack frame as follows:

```
d0-d7/a0-a5 = caller's registers
(a6) = trap handler's static storage
a7 = ssp
8(a7).l = kernel's return program counter
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

The trap handler acts in the same way as a user state trap handler. Note, however, that it is not necessary to restore the caller's **a6** register, as the kernel immediately loads **a6** with the System Globals address. The kernel preserves the caller's **a6** register itself. As with all system state components, the stack used is the calling process's system state stack, in the upper half of the process descriptor.

Prior to OS-9 version 2.3, a system state trap handler was called with a slightly different stack frame:

```
10(a7).l = caller's return program counter
8(a7).w = caller's status register
6(a7).w = exception vector offset
4(a7).w = trap function code
0(a7).l = caller's a6
```

It was the responsibility of the trap handler to restore the caller's **a6** register from the stack frame, add 8 to the stack pointer, and return directly to the caller with an **rte** instruction.

10.4.2 Installing Trap Handlers

A process can install (link to) a trap handler by using the **F\$TLink** system call. This can be explicitly executed in the main body of the program. However, to make the use of trap handlers as transparent as possible, the **F\$TLink** system call can instead be executed automatically when the first **trap #n** instruction tries to call the trap handler. To do this, a program must have an "uninitialized trap handler entry point". This is a routine in the program module, the offset to which is given in the **M\$Excpt** field of the module header. The offset is calculated by the linker, from the entry point symbol given as the seventh parameter to the **psect** directive.

When the program executes a **trap #n** instruction, if the process does not have a trap handler installed for that trap number, the kernel calls the program's uninitialized trap handler routine with the registers and stack frame exactly as for a user state trap handler. The routine should use the vector offset value on the stack to determine which trap handler is required, and execute the **F\$TLink** system call to install the trap handler. It must then re-execute the **trap #n** instruction. This is done by subtracting 4 from the return address in the stack frame, to point again at the **trap #n** instruction, and then returning in the same way as a user state trap handler. This is valid for both user and system state trap handlers. The 'cstart.a' file used for C programs contains such a routine - it is worth studying as an example.

The **F\$TLink** system call attempts to link to the required trap handler. If the trap handler module is not in the module directory, the kernel attempts to load a file of the given name, relative to the current execution directory, and uses the first module in the file. The kernel then allocates and initializes the trap handler's static storage (if any), and calls the initialization routine of the trap handler. The initialization routine of a user state trap handler is called with the following registers and stack frame:

```

d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
a1.l = address of the trap execution routine
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
8(a7).l = caller's return program counter
4(a7).l = zero
0(a7).l = caller's a6 register

```

The initialization routine returns directly to the calling program, not to the kernel, just as the main trap handler execution routine does. The initialization routine must finish by restoring the caller's **a6** register,

removing the 8 bytes of information on the stack, and returning to the calling program with an **rts** instruction.

The initialization routine of a system state trap handler is called with:

```

d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
(a1) = System Globals
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
8(a7).l = return program counter (into kernel)
4(a7).l = zero
0(a7).l = caller's a6 register

```

On return from the system state trap handler's initialization routine, the kernel copies the returned registers **d0-d7/a0-a5**, and **ccr**, to the calling program's register stack frame. The initialization routine must therefore preserve all the registers other than **a6** that its specification does not explicitly state will be changed, just as for a user state trap handler. Note that normally a trap handler preserves all the registers, but OS-9 permits it to return results to the calling program by changing the registers. The system state trap handler's initialization routine is called in system state, as a subroutine of the kernel. It should finish by restoring the caller's **a6** register, removing the 8 bytes of information on the stack, and returning to the kernel with an **rts** instruction. As with all system state components, the stack used is the calling process's system state stack, in the process descriptor.

Prior to OS-9 version 2.3 the initialization routine of a system state trap handler was called with a slightly different set of parameters and stack frame:

```

d0-d7 = caller's registers (d0.w = trap number)
(a0) = caller's trap module name string, updated past end of string
a1.l = the address of the trap execution routine
(a2) = trap handler module header
a3-a5 = caller's registers
(a6) = trap handler's static storage
10(a7).l = return program counter (to calling program)
8(a7) = caller's status register
4(a7).l = zero
0(a7).l = caller's a6 register

```

This initialization routine returns directly to the calling program, not to the kernel, just as a user state trap handler initialization routine does. The initialization routine must finish by restoring the caller's **a6** register, removing the 8 bytes of information on the stack, and returning to the calling program with an **rte** instruction.

10.4.3 Terminating Trap Handlers

Although the module header format for a trap handler includes the offset to a termination routine, the termination routine is never called. The trap handler is simply unlinked and its static storage memory is returned when the program makes the **F\$TLink** system call with a module name pointer of zero (for the appropriate trap number), or when the program terminates.

It is possible that future releases of OS-9 will call the termination routine, so a trap handler should include a termination routine, expecting the same registers and stack frame passed to the initialization routine, and returning to the caller with the carry flag clear. Alternatively, and following Microware's example in the OS-9 Technical Manual, the termination routine could execute an **F\$Exit** system call with a suitable error number (Microware suggest 455).

10.4.4 Writing a Trap Handler in C

The OS-9 Technical Manual gives an example of a user state trap handler in assembly language. However, just as with device drivers and file managers, it is also possible to write a trap handler in C, provided a suitable "skeleton" in assembly language is provided. Such a skeleton (for a system state trap handler) is shown below. Note that by saving the registers on the stack, the skeleton is providing a complete stack frame to the C function that handles the trap instructions, including the condition codes register (**ccr**). From OS-9 version 2.4 onwards, Microware provides the source code of a skeleton user state trap handler, in the directory 'C/SOURCE'.

The trap handler skeleton below presets the **ccr** image to zero, so that the default return is with the carry flag clear, but the C function can set any **ccr** bits (such as the V bit, to indicate arithmetic overflow). Note that in order not to need to reconstruct the stack frame passed by the kernel, the **ccr** image is held in the stack frame location normally used for the **a7** register (**R\$a7** in assembly language, or **a[7]** in C), and is manipulated as a long word (that is, the actual **ccr** image is in the last byte of the four byte field).

```

* File: trapskel.a
* System state trap handler skeleton for a trap handler in C
    use      /dd/defs/oskdefs.d
Typ_Lang    set      (TrapLib<<8)+Objct
Att_Revs    set      (ReEnt+SupStat)<<8
Edition     set      2
    psect    trapskel,Typ_Lang,Att_Revs,Edition,0,TrapEnt
*****
* Static storage (local to trap handler)
*
    vsect
errno:      ds.l      1          standard C error number location
    ends

*****
* Entry point offset table:
*
    dc.l      TrapInit      initialization routine
    dc.l      TrapTerm      termination routine

*****
* TrapInit
* Initialize trap handler
*
* Passed:  d0.w = trap number
*          d1.l = additional static storage allocated (caller's d1)
*          d2-d7 = caller's registers
*          (a1) = trap handler execution entry point
*          (a2) = trap handler module header
*          a3-a5 = caller's registers
*          (a6) = trap handler static storage
*          4(a7) = 0
*          0(a7) = caller's a6 (static storage ptr)
* Returns: carry set if error, with error code in d1.w
* May destroy: ccr
*
* Parameters passed to C function 'trapinit':
*      int trapinit()
* The C function returns zero if no error, else the OS-9 error code.
*
TrapInit
    movem.l  d0-d1/a5,-(a7)  save caller's regs
    move.w   #0,a5          reset stack trace pointer
    bsr      trapinit       call C function
    tst.l    d0             any error?
    beq.s    TrapInit10     ..no
    move.l    d0,4(a7)       overwrite saved d1 with error
    ori      #Carry,ccr      show error
TrapInit10  movem.l  (a7)+,d0-d1/a5-a6  retrieve caller's regs
    addq.l   #4,a7          ditch zero parameter
    rts

```

EXCEPTION HANDLING

* TrapTerm
* Trap handler termination function
* NOTE: at present OS-9 never calls the termination function of a trap handler.
*

TrapTerm
 move.w #1<<8+199,d1 Microware's suggested 'crash'
 os9 F\$Exit

* TrapEnt
* Trap handler main entry point
*
* Passed: d0-d7 = caller's registers
* a0-a5 = caller's registers
* (a6) = trap handler static storage
* 6(a7) = trap vector offset (word)
* 4(a7) = trap function code (word)
* 0(a7) = caller's a6 (static storage ptr)
* Returns: depends on C function 'trapent'
* May destroy: depends on C function
*
* Parameters passed to C function 'trapent':
* void trapent(x,r)
* int x; /* function number */
* REGISTERS *r; /* caller's stack frame ptr */
* The C function may return values to the caller by modifying the stack
* frame (d0-d7/a0-a5 and ccr in R\$a7 only).
*

TrapEnt
 movem.l d0-d7/a0-a5,-(a7) make stack frame
 move.w #0,a5 reset stack trace pointer
 moveq.l #0,d0
 move.w 60(a7),d0 get function code
 move.l a7,d1 copy stack frame address
 clr.l R\$a7(a7) clear "ccr"
 bsr trapent call C function
 movem.l (a7)+,d0-d7/a0-a6 retrieve registers
 move.b 3(a7),ccr set return ccr
 addq.l #4,a7 ditch ccr image
 rts

ends

The main body of the trap handler – written in C – must be in a separate source file. Below is a "do nothing" example, compatible with the skeleton above:

```

/* File: trap.c
   Trap handler main body
*/

#include <errno.h>           /* error numbers */
#include <modes.h>           /* file modes */
#include <types.h>           /* unsigned data types */
#include <MACHINE/reg.h>     /* register stack frame */

/* Static storage: */
int call_count=0;           /* number of calls received */

/* Initialize trap handler */
int trapinit()
{
    return(0);              /* no error */
}

/* Main trap handler function */
void trapent(x,r)
int x;                      /* function number */
REGISTERS *r;              /* caller's stack frame ptr */
{
    call_count++;           /* count calls (for something to do) */
    switch (x) {            /* act on function code */
        case 1:             /* request call count */
            r->d[0]=call_count; /* return call count in d0 */
            break;
        default:            /* unknown request */
            r->d[1]=E_UNKSV;  /* error code */
            r->a[7]=1;        /* set carry flag in ccr */
            break;
    }
}

```

EXCEPTION HANDLING

As when writing a device driver or file manager in C, a **make** file should be used. However, for the purposes of the example, equivalent command lines to assemble, compile, and link the trap handler are shown below:

```
$ r68 trapskel.a -qo=RELS/trapskel.r
$ cc -qr=RELS trap.c
$ 168 RELS/trapskel.r RELS/trap.r -l=/dd/LIB/clibn.l
-l=/dd/LIB/math.l -l=/dd/LIB/sys.l -O=OBJS/trap
```

A simple **make** file to do the same thing is shown below:

```
# make file to make 'trap' module
RDIR = RELS                # directory for ROFs
ODIR = OBJS                # directory for object modules
LDIR = /dd/LIB             # directory for libraries
CFLAGS = -q                # compiler flags for automatically \
                           generated command lines
RFLAGS = -q                # assembler flags for automatically \
                           generated command lines
RFILES = trapskel.r trap.r # names of ROFs

trap: $(RFILES)             # root dependency - make 'trap'
    chd $(RDIR);168 $(RFILES) -l=$(LDIR)/clibn.l -l=$(LDIR)/math.l \
    -l=$(LDIR)/sys.l -O=../$(ODIR)/$@
```

10.5 HARDWARE EXCEPTIONS

These exceptions – such as bus error, address error, and illegal instruction – always occur as a result of a problem in executing a program instruction. The exception may be a normal part of program execution (for example, a **TRAPV** instruction generating an exception as the result of overflow in an arithmetic operation), or it may indicate a programming error (for example, if an illegal instruction exception occurs). Normally, if one of these exceptions occurs during the execution of a process, the kernel will immediately terminate the process, as such an exception implies an unexpected catastrophic error. The exit status of the program is calculated as 100 plus the exception number. For example, a bus error will give an exit status of 102, and an address error will give an exit status of 103. However, in some circumstances the programmer may be able to anticipate and cope with the error. OS-9 therefore provides a means for a program to intercept these exceptions.

Hardware exceptions in system state usually indicate a fatal system error. However, the error may be recoverable or ignorable – preferable to crashing the system. OS-9 therefore provides a separate means for such exceptions to be handled, and the kernel uses this to provide a basic protection against system state exceptions in system calls. If this mechanism has not been reset

or changed by the system call, and a hardware exception occurs during the system call, the kernel returns the exception as an error to the calling program. The error code is 100 plus the exception number. For example, a bus error gives error code 102.

In systems which include one of the Microware ROM-based debuggers, the exception vector table entries for some of the hardware exceptions (bus error, address error, and illegal instruction) point to handlers in the debugger, rather than the corresponding entries in the exception jump table (see the section on the Exception Jump Table). The debuggers have an "enable" command - "e[CR]". If the debugger is "enabled", and one of these exceptions occurs, the debugger is entered and performs a register dump. This allows the programmer to investigate the cause of these potentially fatal error conditions. If the debugger is not "enabled", the debugger jumps to the appropriate entry in the exception jump table, causing the exception to be processed in the normal way by the kernel.

10.5.1 Hardware Exceptions in User State

A process can install handler routines for one or more of the hardware exceptions. The handler routine will only be called if the exception occurs while the process is executing in user state. The handler will not be called if the exception occurs while another process is the current process, or if the exception occurs while the processor is in supervisor state. Once installed, a handler routine may be removed by a further program request, in which case a subsequent exception of that type will cause the program to be terminated.

The handler routines are installed and removed using the **F\$STrap** system call. The program passes a pointer to a list of structures, each describing a handler to be installed. Each structure consists of two 16-bit words. The first word gives the 68000 exception vector offset. For example, the bus error exception is exception number 2, so it has an exception vector *offset* of 8, four times the exception number. Microware have provided symbolic definitions for the exception vector offsets in the file 'DEFS/sysglob.a' (so the symbols are available from the library 'LIB/sys.l', included by the **cc** executive when linking a C program). The symbols all start with the characters **T_**. For example, the bus error exception vector offset has the symbol **T_BusErr**. The following table shows the exceptions that can be intercepted, with their numbers, offsets, and symbolic names for the offsets.

EXCEPTION HANDLING

Description	Number	Offset	Symbol
Bus error	2	8	T_BusErr
Address error - odd address when even required	3	12	T_AddErr
Illegal instruction	4	16	T_Il11Ins
Divide by zero	5	20	T_ZerDiv
CHK instruction	6	24	T_CHK
TRAPV instruction - arithmetic overflow	7	28	T_TRAPV
Privilege violation - supervisor state instruction executed in user state	8	32	T_Priv
Line 1010 emulator	10	40	T_1010
Line 1111 emulator	11	44	T_1111
FPU Branch or set on unordered condition	48	192	T_FPUordC
FPU inexact result	49	196	T_FPIInxact
FPU divide by zero	50	200	T_FPDIVZer
FPU underflow	51	204	T_FPUndrFl
FPU operand error	52	208	T_FPOprErr
FPU overflow	53	212	T_FPOverFl
FPU not a number	54	216	T_FPNotNum

The second word in the structure gives the offset to the routine. The offset is calculated from the end of the structure. The following assembly language line would produce the appropriate structure for a bus error handler function **BusError**:

```
dc.w T_BusErr, BusError-* - 4
```

The list is terminated by a word of -1 (\$FFFF). The list shown below would provide handlers for the bus error, address error, and illegal instruction exceptions:

```
Handlers dc.w T_BusErr, BusError-* - 4
          dc.w T_AddErr, AddError-* - 4
          dc.w T_Il11Ins, Il11Error-* - 4
          dc.w -1
```

The order of the structures in the list is not important (unless there are two for the same exception vector offset!). The calling program passes a pointer to the list in the **a1** register, and a stack frame space pointer in the **a0** register. The kernel saves the handler address and the stack frame space address in two tables in the caller's process descriptor. The handler addresses are saved in the table at **P\$Except**, and the stack frame addresses are saved in the table at **P\$ExStk**.

If the routine offset in a list structure is zero, the kernel clears the handler address in the appropriate entry of the **P\$Except** table. This is the way the handler for an exception can be removed:

```
NoHandlers  dc.w    T_BusErr,0
              dc.w    T_AddErr,0
              dc.w    T_111Ins,0
              dc.w    -1
```

Note that because the offsets are word values relative to the address of the table entries, the handler routines must be located in the program module containing the table, and cannot be more than plus or minus 32k bytes away from the table entry.

The 68020/030 with FPU (68881 or 68882), and the 68040 (which has an internal FPU), can generate additional floating point exceptions. These are supported in the same way by OS-9. The handler addresses and stack frame addresses are saved in additional tables in the process descriptor, **P\$FPExcept** and **P\$FPExStk** respectively.

When a hardware exception occurs in user state, the kernel uses the vector offset of the exception as an index into the table in the process descriptor of the current process. If the handler address is zero, the kernel terminates the process, giving it an exit status of 100 plus the exception number. Otherwise, the kernel builds a register stack frame in the memory whose address was given by the **F\$STrap** system call. If the stack frame space address is zero, the kernel uses the process's current user stack pointer.

Note that in either case the kernel builds the stack frame *below* the address given (simulating a push down stack). Therefore when giving a fixed address at which to build the stack frame, the program must add the size of the stack frame to the base address of the storage before passing it to the **F\$STrap** system call. The size used by the kernel is calculated as **R\$Size-2** (70 bytes) – that is, no exception format and vector word is written to the stack frame. Before building the stack frame, the kernel checks that the user has write permission for the memory. If not, the process is terminated, with a stack overflow exit status (**E\$StkOvf**).

The kernel then pushes the status register at exception and the address of the handler onto the system state stack, and so calls the handler by executing an **rte** instruction. The exception handler is called in user state (the state at the time of exception). It does not return to the kernel. The effect is as if the program had built the stack frame and jumped (not a subroutine call) to the handler routine, instead of executing the instruction that caused the exception.

EXCEPTION HANDLING

The handler routine is passed:

- d7.1 = exception vector offset (exception number times 4)
- a0.1 = program counter at exception
- a1.1 = user stack pointer at exception
- (a5) = register stack frame
- a7.1 = a5.1 unless explicit stack frame space specified
- 66(a5) = program counter at exception (= a0)
- 64(a5) = status register at exception
- 60(a5) = user stack pointer at exception (= a1)
- 0(a5) = d0-d7/a0-a6 at exception

The exception handler is effectively jumped to as a change of flow of control in the program. It must decide whether and how to continue execution of the program. It may decide that it can fix the problem, and allow the main program to continue execution. Having fixed the problem, the exception handler would restore the program's registers from the stack frame (including the condition codes register - **ccr**), restore the program's stack pointer (passed to the exception handler in the **a1** register), and then jump back to the program, using the program counter passed in the **a0** register. Note that the program counter will not normally point at the instruction that caused the exception. Usually it will have been incremented by the processor to point at the next instruction, but for exceptions caused by a memory access (bus error and address error) the program counter may point part of the way through the instruction that caused the exception.

Alternatively, the exception handler may decide to continue execution at a different point in the program, or to terminate the program (perhaps preceded by a "clean up" sequence). Just as with a signal handler routine, the exception handler can execute any system calls - it is executing as a part of the program, in user state - but because it is called asynchronously, it must be careful not to use program variables that may be in use by the main body of the program.

The main concepts to understand in order to write a user state exception handler under OS-9 are:

- The exception handler is effectively asynchronously jumped to (not a subroutine call).
- The kernel builds a register stack frame *below* the given memory address, or on the user stack if no address was specified. The stack frame contains the data and address registers (including the stack pointer), the status register, and the program counter, as they were at exception.

10.5.2 Example – Bus Error Handler

The "bus error" exception is probably the exception most commonly required to be intercepted by a program. A bus error exception occurs if the processor "bus error" input signal is asserted in response to a memory access, instead of the normal termination of a memory access. External circuitry on the processor board normally asserts the bus error input if a memory access attempted by the processor has not completed within a certain time, indicating that no device is responding to the address that has been put out by the processor.

The timeout depends on the processor board (and some boards have a programmable timeout), but a time of the order of 200 microseconds is typical. The timeout may occur because the address does not match the address of any device (memory chip or I/O interface chip) in the system, or because the device is currently busy, and refuses to respond. The bus error signal is also asserted by the Memory Management Unit (MMU) if one is in use (by the System Security Module, under OS-9), and a program tries to access a memory location that is not within its current memory map, or it tries to write to a location for which it does not have write permission.

A program that checks for the existence of an area of memory, or an I/O interface, will need to install a bus error exception handler, to handle the exception that will occur if the memory or interface chip is not present. Also, a program that directly accesses an I/O interface that is sometimes busy will need to install a bus error exception handler to retry an access to the interface. In the first case the program will not want to retry the instruction that caused the bus error – it will set a "device does not exist" flag. In the second case the program will want to retry the instruction, perhaps with a maximum number of attempts. This can be done by resetting the program counter to point again at the instruction, or by setting a flag and jumping to the end of a loop in the program to test for success or failure.

The example below shows a bus error exception handler for the first case. The program is attempting to determine whether an I/O interface chip is present at the given address in this system. The main body of the program is in C, but the function to make the **F\$STrap** system call must be in assembly language, as must the exception handler (or at least a skeleton function). In this example the assembly language function **probe_byte** attempts to read a byte from the given memory address. If it succeeds (no bus error), the function finishes in the normal way, returning the **d0** register set to zero. Otherwise, the exception handler is called, which sets the **d0** register to -1,

EXCEPTION HANDLING

and jumps to the instruction in the **probe_byte** function following the instruction to set **d0** to zero.

```
#include <stdio.h>
#include <errno.h>
#include <MACHINE/reg.h>

#define ERROR (-1)

REGISTERS stack_frame;          /* structure for stack frame */

int f_strap(),probe_byte();     /* declare functions */

main(argc,argv)
int argc;
char **argv;
{
    char *check_addr;

    /* The address to test is a command line parameter: */
    if (argc!=2 || sscanf(argv[1],"%x",&check_addr)!=1)
        exit(_errmsg(1,"Invalid board address\n"));
    if (f_strap(&stack_frame)==ERROR) /* install handler */
        exit(_errmsg(errno,"Can't install handler\n"));
    /* Test for the existence of a device at the address given: */
    if (probe_byte(check_addr)==-1)
        _errmsg(1,"No board at address %08x\n",check_addr);
    else
        _errmsg(1,"Board exists at address %08x\n",check_addr);
}
/* Function to install bus error handler
   Passed:      address of stack to use (zero to use program stack)
*/
#asm
f_strap:    movem.l  d1/a0-a1,-(a7)    save registers
            lea      ExcpTbl(pc),a1   point at table of handlers
            tst.l    d0               any stack given?
            beq.s    f_strap10        ..no
            addi.l    #R$Size-2,d0    convert to pointer to top of stack
f_strap10   movea.l    d0,a0           copy top of stack address
            os9       F$STrap         make the system call
            bcs.s    f_strap20        ..error
            moveq     #0,d0           show no error
            bra.s     f_strap30        ..and return
f_strap20   move.l     d1,errno(a6)    save error code
            moveq     #-1,d0          show error occurred
f_strap30   movem.l    (a7)+,d1/a0-a1  retrieve registers
            rts                    return to C program
* Table of exceptions to handle, and handler offsets:
ExcpTbl     dc.w      T_BusErr,bus_hand*-4
            dc.w      -1              end of table marker
```

```

* Read a byte from a specified address:
* Passed:  d0.l = address to test
probe_byte: move.l  a0,-(a7)      save register
            move.l  d0,a0        copy address to use
            move.b  (a0),d0      read byte
* The following instruction is only executed if no
* bus error occurred:
            moveq   #0,d0        show no bus error

probe_byte10
            movea.l (a7)+,a0      retrieve register
            rts               return to C program

* Bus error handler:
bus_hand:  movea.l  a1,a7        restore stack pointer
            movem.l (a5),d0-d7/a0-a4  restore regs from stack frame
            movea.l R$a5(a5),a5  restore a5 from stack frame
            moveq   #-1,d0       show bus error occurred
            bra.s   probe_byte10 ..finish off

#endasm

```

In the more generalized case, where the bus error exception could occur in more than one place (for example, separate functions might be used to try reading a byte, or a word, or a long word), the program could set a static storage variable to indicate which function is executing, or the program could save in static storage the program address at which to continue execution after a bus error.

10.5.3 'move from sr' and 'move from ccr'

The **move from sr** (copy the status register) instruction is not a privileged instruction on the 68000/010, and these members of the 68000 family do not have a separate **move from ccr** (copy the condition codes register) instruction. In contrast, the higher members of the 68000 family (68020/030/040) have a **move from ccr** instruction, and on these processors the **move from sr** instruction is privileged – a "privilege" exception occurs if this instruction is executed in user state.

In order to be compatible with both groups of processors, the kernel checks the "illegal instruction" and "privilege" exceptions, to see if they are due to a **move from ccr** or **move from sr** instruction respectively. If so, the kernel emulates the instruction, by moving the **ccr** register to the destination specified in the instruction, rather than passing the exception to the process's exception handler (or terminating the process if it has no handler). This makes programs written with either instruction execute correctly on both groups of processors.

10.5.4 Hardware Exceptions in System State

The processor is in supervisor state during the execution of a system call, a system state program, a system state trap handler, or an interrupt service routine. When a hardware exception occurs in system state, the kernel handles the exception in a different manner from a hardware exception in user state.

A hardware exception during interrupt handling (which is always in system state) is considered a special case. If the hardware exception occurs during an interrupt service routine (the **D_IRQFlag** field in the System Globals is not negative), the operating system gives up through a controlled system crash. It prints a "System state exception" message on the system console (reporting the exception vector offset), and attempts a soft reset (jump to the bootstrap ROM entry point). If a ROM-based debugger is available, the kernel calls the debugger, rather than jumping straight to a soft reset. The kernel "crashes" the system because it cannot know whether the interrupt has been successfully handled, or whether the I/O device is now in a non-functional state, with further use possibly resulting in a corruption of a filing system.

Therefore if an interrupt service routine anticipates that it may cause a hardware exception, it should temporarily patch the exception jump table before executing the instruction that may cause the exception. This is a perfectly valid mechanism – it cannot cause conflict, because interrupts are handled in a purely hierarchical prioritized order (this is a function of the processor) – they cannot sleep or be switched out.

If the processor is not executing an interrupt handler at the time of the system state exception, the kernel attempts to call a system state exception handler for the current process. Like the user state exception handlers, the address of the exception handler and the address of the stack to use are held in the process descriptor, so separate exception handlers can be set for each process. This allows a process to go to sleep without having to save and restore the handler address and stack pointer. Unlike the user state exception handlers, there is only one system state hardware exception handler address field in the process descriptor (**P\$ExcpPC**) for handling all system state hardware exceptions, and only one field for the stack pointer to use at exception (**P\$ExcpSP**).

If the stack pointer field (**P\$ExcpSP**) of the process descriptor of the current process is zero the kernel gives up through a controlled system crash, as described above. This is the default case while executing a system-state process or system-state trap handler. Otherwise the kernel loads the **a7** register (the stack pointer) with the value in the **P\$ExcpSP** field of the

process descriptor, unmask all interrupts, and jumps to the address given in the **P\$ExcpPC** field.

When called to execute a system call (**trap #0** instruction) the kernel installs a default exception handler, which simply returns the exception as an error to the caller. The error code is the exception number plus 100. The kernel restores the original handler (usually "none") before returning to the caller.

However, the **I\$Attach** system call routine temporarily installs its own exception handler before calling the initialization routine of the device driver, such that an exception is treated as an initialization error (returning an error code of 100 plus the exception number). This causes a program to get an error **E_BUSERR** if it attempts to open a path to a device for which the hardware is not present. In addition, the device driver has the opportunity to de-allocate any allocated resources, because its termination routine is called by the **I\$Attach** system call, as happens if the initialization routine returns an error in the normal way.

Any system state routine can intercept hardware exceptions by temporarily replacing the stack pointer and exception handler fields in the process descriptor of the current process. On exception, the kernel jumps to the exception handler address, effectively causing an asynchronous change of flow of control in the system state routine that caused the exception. The handler is called as follows:

```
d1.w = exception number plus 100
d2-d6/a1-a3/a5 = registers at exception
d7.l = exception vector offset
(a4) = current process's Process Descriptor
(a6) = System Globals
a7.l = value taken from P$ExcpSP
sr = interrupt mask is clear
```

Registers **d0-d1/d7/a0/a4/a6** and **ccr** are lost. Note that the kernel does not place any parameters on the stack. The example below shows a system state routine recovering from a bus error:

```
* The address of the location to test is passed in the d0 register:
ProbeByte:  movem.l d1-d2/a0,-(a)      save registers
            move.l P$ExcpPC(a4),-(a7)  save current values
            move.l P$ExcpSP(a4),-(a7)
            lea    ProbeByte10(pc),a0  build recovery PC
            move.l a0,P$ExcpPC(a4)     set recovery PC
            move.l a7,P$ExcpSP(a4)     and stack pointer
            movea.l d0,a0               copy the address to test
            moveq  #-1,d2               default to bus error occurred
            tst.b  (a0)                 test the memory location
            * The next instruction is only executed if no exception occurred:
```

EXCEPTION HANDLING

```

        moveq    #0,d2                no bus error
ProbeByte10 move.l (a7)+,P$ExcpSP(a4)  restore old values
        move.l   (a7)+,P$ExcpPC(a4)
        move.l   d2,d0                copy the result
        movem.l  (a7)+,d1-d2/a0       retrieve registers
* The d0 register now contains 0 if no bus error (or other hardware
* exception) occurred, otherwise it contains -1.
        rts
```

10.6 INTERRUPTS

10.6.1 How 68000 Interrupts Work

An interrupt is an external signal to the processor requesting the asynchronous execution of a subroutine, known as an interrupt handler. In general, interrupts are generated by I/O interface chips when they require servicing by the processor – for example, when a serial port interface has received a character. The 68000 family processors respond to an interrupt by an exception, allowing the interrupt handler to be executed in supervisor state, and afterwards the interrupted program to continue execution. These processors do not provide just a single interrupt input signal. Instead, they have a 3-bit binary coded input. This is generated by an external priority encoder chip, that takes 7 interrupt inputs (numbered 1 to 7), and outputs the 3-bit binary value indicating the number of the highest active input. If no input is active, the priority encoder generates a code of zero, meaning no interrupt handling is currently required.

This mechanism provides a prioritized system of seven levels of interrupts. If the input interrupt code exceeds the current interrupt mask value in the processor's status register, the processor initiates exception processing of the interrupt, with a special memory access cycle known as an interrupt acknowledge cycle. The processor (having saved the current status register, as in all exceptions), also sets the interrupt mask in the status register to equal the level of the interrupt being serviced. Thus, until the interrupt handler finishes, any other interrupt on the same or a lower level is ignored, but a higher level interrupt can cause a further exception, interrupting the interrupt handler of the lower level interrupt. Note that it requires a privileged instruction to change the interrupt mask in the status register (as with any part of the high byte of the status register word), so user state programs cannot mask interrupts.

Interrupt level 7 is a special case. Setting the interrupt mask to 7 does not prevent the processor responding to a level 7 interrupt, making such interrupts "non-maskable". Note, however, that because the processor only

starts interrupt processing if the interrupt level goes above the interrupt mask, or the interrupt mask is lowered below the current interrupt level, if the interrupt handler completes without clearing the level 7 interrupt, and the interrupt mask restored by the `rte` instruction at the end of the handler is 7, a further exception is not taken.

The processor must have some means of determining which device caused the interrupt, because most systems will have more than one device generating an interrupt. Simple processors require that software poll the status register of all devices that may be interrupting, to see which is currently generating an interrupt. However, the 68000 family processors can take distinct exceptions for different interrupt sources, by using separate interrupt exception vectors. These processors support two methods by which the interrupt vector is generated.

The first method is known as normal vectoring. The device (or some associated circuit) that is generating the interrupt responds to the interrupt acknowledge cycle by returning a vector number. The second method is known as auto-vectoring. This allows the use of devices that cannot themselves return a vector number. External circuitry detects that the interrupting device is of this type, and asserts an "auto-vector" input signal to the processor in response to the interrupt acknowledge cycle. The processor then generates the vector number internally, by adding 24 to the level of the interrupt. For example, a level 3 auto-vector interrupt generates a vector of 27.

10.6.2 Using Interrupts Under OS-9

For compatibility with all members of the 68000 family, and with most I/O devices, vector numbers are limited to 8 bits. Most devices will allow any vector number to be programmed into their interrupt vector register, to be used in response to a future interrupt acknowledge cycle. However, Motorola have reserved vector numbers 0 to 63 for other types of exception (including auto-vectored interrupts). Therefore 192 vector numbers are available for normal vectored interrupts, and 7 for auto-vectored interrupts, making a total of 199 interrupt vector numbers.

Most systems will use only a few of these vectors, and some will use the same vector for more than one device. This is particularly true of auto-vectored interrupts, as only 6 maskable levels are available, but it should be avoided with normal vectored interrupts. Therefore OS-9 provides a simple mechanism to allow any number of interrupt handlers to be installed on any

EXCEPTION HANDLING

number of vectors, without absolutely requiring that any handlers be installed at all (the kernel has a default handler for unexpected interrupts).

In accordance with the OS-9 philosophy of dynamic configurability, interrupt handlers are installed when needed, and removed when no longer required. The **F\$IRQ** privileged system call is used to install an interrupt handler. The caller passes the address of the handler, the address of the static storage to be used by the handler, a "port address", the interrupt vector number, and a software polling priority value. Usually, an interrupt handler is part of a device driver. In this case, the static storage is normally the Device Static Storage, the "port address" is normally the base address of the registers of the interface chip, and the vector number and polling priority are taken by the device driver from the device descriptor.

The kernel maintains an "interrupt polling table". This is an array of structures, initially all free, which are used to link interrupt handlers to interrupt vectors. The size of the table – which is not dynamically expandable – is determined by an entry in the **init** configuration module. The **F\$IRQ** system call searches for a free entry in this table, and stores the parameters there. The kernel then uses the vector number to select one of 199 pointers in the System Globals. This pointer is the root of a linked list of polling table entries for that vector number. The kernel then searches the linked list, which is sorted by the software polling priority value – a low value means the entry is placed nearer the start of the list. It inserts the new entry after all entries with a lower or equal software polling priority. If the root pointer was null (zero), the kernel knows that the linked list for the given vector number was empty, and makes the new entry the first entry in the linked list, placing its address in the root pointer.

A software polling priority of zero is a special case. It is used to ensure that the handler is the only handler on the given vector number. If there is already a handler installed (the root pointer is not null), the caller is returned an error – **E\$VctBsy**. If an **F\$IRQ** system call attempts to install a handler on a vector which already has an entry of software polling priority zero, it is returned the same error. This mechanism is necessary to ensure correct support for some devices that have no status flag to indicate that they are generating an interrupt. The only way of knowing that it is this device that is interrupting is by the unique vector number returned by the interrupt acknowledge cycle.

The **F\$IRQ** system call is also used to remove an interrupt handler from the polling table. The caller passes zero in place of the interrupt handler address. The kernel again uses the given vector number to identify the appropriate

root pointer. It then searches the linked list for that vector until it finds an entry whose static storage pointer matches that passed to the **F\$IRQ** routine. Having found the correct entry, the kernel unlinks it from the linked list, and marks it as free for use by a subsequent **F\$IRQ** call to install a new handler. This has the corollary effect that two interrupt handlers must not be installed on the same vector with the same static storage address. However, this is not a restriction, as there are almost no circumstances imaginable where a programmer would wish to do this.

Also note that two devices using different interrupt levels should not use the same interrupt vector. Otherwise, as the kernel calls each handler in the linked list for the vector in turn, an interrupt handler could be called recursively.

The interrupt polling table structure is described in more detail in the chapter on the OS-9 Internal Structure.

The kernel has a single "core" interrupt handler, and the exception jump table entries (see below) for all of the interrupt exceptions jump to this interrupt handler. The core handler uses the vector offset pushed on the stack by the jump table entry to select the appropriate root pointer for this interrupt exception number. It then calls each handler in the linked list in turn, passing the static storage and port addresses as specified in the **F\$IRQ** call, until a handler returns the carry flag clear, indicating that the handler has recognized and serviced the interrupt. Finally, the kernel returns from the exception, using the **rte** instruction. Note that if the interrupt occurred while the processor was executing in user state, the kernel first performs its "return to user state" checks on the current process, such as whether the process is now marked as "timed out". This permits functions such as task switching after a clock tick interrupt, and the calling of the process's signal handler routine if the process is sent a signal by an interrupt handler.

If there is no handler for the interrupt vector (the root pointer is null), or all handlers on the vector return the carry flag set, the kernel increments the byte field **D_UnkIRQ** ("unknown interrupt request") in the System Globals, and then returns from the exception. If the interrupt persists the kernel's interrupt handler will be called again, and the count will eventually roll over to zero (after 256 attempts). When this happens the kernel masks interrupts up to the level of the offending interrupt, preventing the processor from responding to the interrupt again. The **D_UnkIRQ** field is cleared whenever any interrupt is successfully processed. This is a measure to protect against hardware glitches in the external interrupt circuitry - normally an unrecognized interrupt is fatal for any system.

EXCEPTION HANDLING

Note that unless bit zero of the first compatibility byte in the **init** configuration module is set, the kernel only saves the registers **d0-d1/a0/a2-a3/a6** on interrupt, to speed up the response to the interrupt. Therefore interrupt handlers that use other registers must save and restore them. The current version of the C compiler generates code that preserves all of the data and address registers not preserved by the kernel, including any floating point unit (FPU) data registers. However, if the interrupt handler does use the FPU, it must save and restore the FPU context, as an interrupt can break into an FPU instruction:

IRQSvc	tst.b	D_68881(a6)	does the system have an FPU?
	beq.s	IRQSvc10	..no
	fsave	-(a7)	save FPU context
	fmovem.l	fpcr/fpsr/fpiar, -(a7)	save FPU control registers
IRQSvc10	bsr	IRQSvcMain	service the interrupt
	move	sr, d0	save carry flag
	tst.b	D_68881(a6)	does the system have an FPU?
	beq.s	IRQSvc20	..no
	fmovem.l	(a7)+, fpcr/fpsr/fpiar	restore control registers
	frestore	(a7)+	restore FPU context
IRQSvc20	move	d0, sr	restore carry flag
	rts		

An interrupt handler terminates with an **rts** instruction, not an **rte** instruction. This is because the handler is returning to the kernel's core interrupt handler, which itself executes the **rte** instruction to finish the exception processing.

During interrupt processing the kernel switches to a different stack – the interrupt stack – to avoid the need for stack to be reserved for interrupt processing in the system state stack of every process descriptor. The size of the interrupt stack (which is not dynamically expandable) is specified in the **init** configuration module. The kernel's interrupt handler saves the system stack pointer, and then increments the **D_IRQFlag** field of the System Globals. This field is initialized to -1 during the kernel's coldstart. If it is now zero, the kernel knows that this interrupt is not breaking into the service of another interrupt (which will have already switched to the interrupt stack), so it switches to the interrupt stack by loading the **a7** register from the **D_SysStk** field of the System Globals. At the end of the interrupt service the kernel decrements the **D_IRQFlag** field, and restores the original stack pointer.

Because an unrecognized (and therefore unserviced) interrupt is potentially fatal for the system, interrupt handlers must not be subroutines in modules that can be unexpectedly terminated (such as trap handlers, and user state programs), and must not use static storage that can be unexpectedly

de-allocated (such as program or trap handler static storage). Therefore only operating system components should contain and install interrupt handlers. The use of interrupts within device drivers is explained in the section on "Device Drivers".

10.6.3 Interrupts OS-9 Cannot Handle

Because OS-9 only maintains root pointers for the 199 normally expected interrupt exception vectors, there are two types of interrupt exception that OS-9 cannot handle. The first is the case in which a device returns an interrupt vector (in response to an interrupt acknowledge cycle) that is not in the range 64 to 255. This should never happen. It indicates that there is a hardware fault, or that the device has been programmed with an improper vector by the device driver. If the vector corresponds to the vector for a different type of exception (such as an illegal instruction), the kernel will act as if that exception had occurred - it has no way of knowing that in fact an interrupt generated the exception.

If the exception occurred in user state and the vector does not match any known exception vector, the kernel kills the current process, giving it an exit status of 100 plus the exception vector number. If the exception occurred in system state the kernel treats it like a normal "hardware" exception in system state (see above).

This generation of an invalid vector may happen for certain devices that, on reset, set their interrupt exception vector register to a value of 15 - the 68000 "uninitialized interrupt" vector. If such a chip is then programmed to generate an interrupt without first writing a valid vector number to its interrupt exception vector register, OS-9 will be unable to handle the interrupt.

The second type of interrupt that OS-9 cannot handle is the Spurious Interrupt exception (vector 24). This exception is taken by the processor if the Bus Error input signal is asserted in response to the interrupt acknowledge cycle. Usually this is because no device has responded to the interrupt acknowledge cycle, so the memory access timeout circuit on the processor board asserts the Bus Error signal. This happens on VME systems and other bus based systems that use a daisy-chained interrupt acknowledge signals if the backplane jumper that by-passes the daisy chain for a particular backplane slot is left out when the slot is empty. If a board further down the backplane generates an interrupt, it will not receive the interrupt acknowledge signal from the processor board (because the daisy chain is broken by the empty slot and the missing jumper), and so will not respond.

EXCEPTION HANDLING

The timeout circuit on the processor board eventually times out, and asserts the Bus Error signal.

Although OS-9 has a root pointer in the System Globals corresponding to the Spurious Interrupt exception vector (which is equivalent to an auto-vector of level zero - 24), the **F\$IRQ** system call will not permit a handler to be installed on this vector, and the exception jump table entry for this exception jumps to the kernel's "hardware" exception handler. The kernel handles the exception as described above for invalid interrupt vectors.

However, the interrupt from the interrupting device has not been acknowledged or serviced, so the interrupt signal remains asserted, and a further exception is taken once the interrupt mask is cleared (either explicitly by the kernel in its handler for hardware exceptions in system state, or implicitly by the **rte** "return from exception" instruction). Therefore the system will "hang", or - if the interrupt occurred (at a higher level) while another interrupt was being serviced - the kernel will give up through a controlled system crash. In the latter case the exception is recognizable because the system state exception message reports the exception vector offset for a Spurious Interrupt exception - \$0060.

10.6.4 The Level 7 Interrupt

As mentioned above, a level 7 interrupt is non-maskable. Therefore it should never be used for normal interrupt handling, as the main body of the module (such as a device driver) initiating the interrupt cannot mask interrupts while manipulating variables or device registers also used by the interrupt handler.

Similarly, a level 7 interrupt handler must not make any system calls, as the kernel cannot protect its data structures by masking the interrupt.

If one of the Microware ROM-based debuggers is installed, the vector for auto-vector level 7 in the exception vector table is set to point to a handler in the debugger, rather than to the kernel's core interrupt handler. Many processor boards provide a front panel "abort" switch that generates a level 7 auto-vector, so this facility can be used to call the ROM-based debugger in the event that the system "hangs", in order to try to determine why the "hang" occurred. Therefore, if the level 7 auto-vector is to be used for some other purpose (such as processor emulation of DMA), the exception vector table entry for auto-vector level 7 must be overwritten, unless it is certain that no ROM-based debugger will be used.

10.7 THE EXCEPTION VECTOR TABLE

The processor selects which handler routine to call on exception by using the exception number as an index into a table of addresses, known as the exception vector table. Because there are 256 possible vector numbers, the table is 256 long words in length – that is, 1k bytes. The first entry (corresponding to vector zero) is reserved for the address to load into the stack pointer on reset, and is used in OS-9 to point to the System Globals. The 68000 processor always locates the exception vector table at address zero, while the 68020/030/040 have a vector base register (**vbr**) that gives the address of the table. The **vbr** is set to zero when the processor is reset, but may be re-programmed by the bootstrap ROM (the OS-9 kernel does not write to this register).

The reset vector and reset stack pointer values must be in ROM, so that they are present on power-on. However, it is also convenient to have the exception vector table in RAM, so that it can be dynamically modified. Processor boards address this dichotomy in a number of different ways:

- a) The exception vector table is in ROM, at address zero, and cannot be modified. It is part of the bootstrap ROM. The system RAM starts at some other address.
- b) The system RAM is mapped to start at address zero, but the first two long words are overlaid by a reflection of the first two long words of the ROM. The reset vector and reset stack pointer are fixed, but the other vectors are dynamically modifiable.
- c) The system RAM is mapped to start at address zero, but is overlaid by the ROM on reset. Either the termination of the reset cycle, or an explicit processor board register write, causes the ROM reflection to disappear. The vectors are then dynamically modifiable.
- d) The ROM is mapped to start at address zero, and contains the reset vector and reset stack pointer. However, the processor contains a vector base register, so the exception vector table can be relocated to any part of the system RAM.

If the exception vector table is in ROM only, it forms the first part of the bootstrap ROM, and cannot be modified. Otherwise, the bootstrap program builds the exception vector table in RAM. Each entry in the exception vector table (except the reset stack pointer and reset vector) points to the corresponding entry in the exception jump table (see below). However, as

described above, if the bootstrap ROM contains one of the ROM-based debuggers, then the bus error, address error, illegal instruction, and auto-vector level 7 interrupt exception vectors point to handlers in the debugger. Because the kernel must support all types of system, it does not attempt to modify the exception vector table. Instead, it assumes that the table entries point to the appropriate entries in the exception jump table, and writes the addresses of its handlers in the exception jump table.

If it is known that the exception vector table is in RAM, a vector may be overwritten to cause the exception to call a user-installed handler directly. This by-passes all the kernel mechanisms (such as the use of the interrupt stack) and protections, and so is not recommended by Microware, but gives a slightly faster interrupt response that may be necessary in some critical applications.

10.8 THE EXCEPTION JUMP TABLE

The exception jump table is needed for two reasons. Firstly, the 68000 does not save a record of which exception is being serviced. Although the higher members of the family do (the exception processing pushes the exception number on the stack), OS-9 must be compatible with all members of the family. Secondly, the exception vector table may be in ROM, so the vector addresses cannot be modified, yet the bootstrap ROM cannot know the addresses of the kernel's exception handlers. Therefore each exception vector points to its own entry in the exception jump table. Each entry in the exception jump table consists of an instruction to push the exception vector offset onto the stack, followed by an absolute jump instruction, which jumps to the exception handler. The first two entries in the exception vector table (reset stack pointer, and reset vector) do not need corresponding exception jump table entries, so the first entry is for exception number 2 (bus error). Therefore if the exception jump table is disassembled, the first few instructions look like this:

```
pea    $0008.w
jmp    $xxxxxxx.l
pea    $000C.w
jmp    $yyyyyyy.l
pea    $0010.w
jmp    $zzzzzzz.l
```

The kernel builds the exception jump table as part of its coldstart routine. The entries in the table are each 10 bytes, so the table is 2540 bytes in size. Its start address is usually 4k below the address of the System Globals, but to

allow for variations in the size of this memory area the kernel writes the jump table base address to the **D_ExcJump** field of the System Globals¹².

The exception jump table is always in RAM, so an operating system component that wishes to permanently or temporarily redirect an exception to its own handler can overwrite the jump address (the last 4 bytes) in the corresponding jump table entry. If the change is to be temporary, the routine should save the current address, set its own address, perform the function that might generate the exception, and then replace the original address of the kernel's handler. This strategy also allows this method of redirection to be nested. For example, a high level interrupt handler could temporarily redirect the bus error vector, even though this interrupted a similar attempt by a low level interrupt handler. The example below temporarily replaces the bus error handler while accessing an I/O device register:

```

        move.l  D_ExcJump(a6),a0      get address of jump table
* Each entry is 10 bytes, and the jump address is the last long word
* of the entry. The first entry is for exception 2. Bus error is
* exception 2:
        adda.w  (2-2)*10+6,a0        point at jump address for bus error
        move.l  (a0),-(a7)           save the jump address
        lea     BusError(pc),a1      point at our handler
        move.l  a1,(a0)              set the new jump address
        move.l  a7,d1                save the stack pointer
        moveq   #-1,d0               default to "bus error occurred"
        move.w  (a3),d2              the instruction that may cause a
*                                   bus error
* The next instruction is only executed if no bus error occurred:
        moveq   #0,d0                show "no bus error"
BusError  movea.l d1,a7               restore the stack pointer
        move.l  (a7)+,(a0)           restore the original jump address
        tst.l   d0                   did a bus error occur?
```

As with overwriting the exception vector table, this mechanism should only be used when strictly necessary, but it is available when needed.

¹² The 'CBOOT' and **ROMBug** options for the boot program from OS-9 version 2.4 onwards may allocate static storage that increases this memory area above 4k bytes.

CHAPTER 11

OS-9 SYSTEM CALLS



All operating system functions are accessed by means of system calls. A system call is essentially a subroutine that is executed in system state. It is important to appreciate that the system call is effectively a subroutine call by the calling program, so that all operations performed by the system call handler function are performed on behalf of the calling process. For example, if a system call function goes to sleep, it is the calling process that is sleeping. Similarly, the calling program will not continue execution until the system call is completed, just as would be the case with a simple subroutine call.

Conversely, the operating system does nothing without a system call being made by a program. The only time that operating system code executes other than when servicing a system call is when servicing an interrupt.

11.1 THE SYSTEM CALL MECHANISM

OS-9 system calls use the 68000 family **trap #0** instruction. This instruction – an extension of the software interrupt of earlier processors – causes a processor exception. The processor saves the status register and program counter, switches to supervisor state, and continues execution at the address indicated by the appropriate exception vector. There are 16 **trap** instructions, **trap #0** to **trap #15**. OS-9 uses **trap #0** for system calls.

The required system function is indicated by the word following the **trap #0** instruction in the program. The kernel reads this word to determine the required function, using it as an index into the appropriate Dispatch Table (System or User) depending on whether the caller was in system or user state. The kernel then adds two to the saved program counter, so that

program execution will continue with the instruction following the function code word.

The OS-9 assemblers **r68** and **r68020** provide a built-in macro **os9** to generate the **trap #0** instruction and the function code word in one statement. The function codes are defined symbolically in the file 'DEFS/funcs.a', and so are available as external references resolved by the library 'LIB/sys.l'. For example, the "fork a process" system call has function code 3, defined symbolically as **F\$Fork**. So the assembly language statement:

```
os9      F$Fork
```

is equivalent to the statements:

```
trap     #0
dc.w     F$Fork
```

or:

```
trap     #0
dc.w     3
```

The **trap** instructions, as with any exception, put the processor in supervisor state, which is the state required by the operating system. Note that the 68000 family processors can only go from user state to supervisor state as the result of an exception.

11.2 SYSTEM CALL PARAMETERS

The **trap #0** exception handler in the kernel saves all of the caller's registers on the system stack¹³ as a "stack frame", and retrieves them before returning to the caller. Therefore the caller's registers are not modified unless explicitly stated in the documentation, because the system call function must actively change the stack frame to affect the caller's registers.

OS-9 was originally written exclusively in assembly language (although now some parts are written in C), with speed of execution in mind. Therefore all parameters to system calls, and all returned values, are in processor registers, or are pointed to by processor registers. Which registers hold which values is determined by the specification for the particular system call.

The error returning convention is the same for all system calls, and for all communications between operating system components. The kernel returns an error to the calling process in the caller's registers – the "carry" flag of the caller's condition codes register (**ccr**) is set, and the error code is placed in

¹³ The system stack is in the second half of the calling process's process descriptor, except during interrupt processing.

the low word (bits 0–15) of the **d1** register. If there is no error, the carry flag is cleared, and the **d1** register is not modified.

The passing of parameters and results in the processor registers and the setting of the carry flag to indicate an error are techniques that are not directly compatible with C. Therefore it is necessary to provide small library functions written in assembly language that can be called from C programs. Such functions translate the C parameter passing format to that required by the system call, make the system call, and then convert the returned values and error indication to a form compatible with C. This topic is covered in detail in the chapter on Microware C and Assembly Language.

11.3 CUSTOM SYSTEM CALLS

System state modules can add new system calls, or replace existing ones, using the **F\$\$\$Svc** system call. This makes OS-9 almost infinitely customizable. Note that this is a privileged system call – normal user state programs cannot add or modify system calls.

On coldstart the kernel installs its own system call handlers in the Dispatch Tables, using the **F\$\$\$Svc** system call routine. The kernel then tries to link to one or more "kernel customization modules", whose names are given in the **init** module. If found, the kernel calls their entry points, giving them a chance to allocate memory, set up data structures, and install system calls using **F\$\$\$Svc**. The system programmer can therefore add or modify system calls without altering the kernel.

New system calls can make themselves extensions to existing system calls by saving the address of the existing routine (from the Dispatch Tables) before installing the new routine. When called, the new routine performs its additional function, and then jumps to the old routine (or calls the old routine first, as appropriate).

The **F\$\$\$Svc** system call optionally instructs the kernel to save a memory address in the Dispatch Table with each system call handler address. A kernel customization module can include static storage definitions, in the same way as a program. If it does, the **M\$Mem** field of its module header gives the total size of the required static storage. Before calling the initialization routine of a kernel customization module, the kernel allocates an area of memory of the required size, and passes its address to the customization module's initialization routine in the **a3** register.

The **FSSvc** system call expects the "memory address" parameter to be in the **a3** register, so when the initialization routine makes this system call to install a new or replacement system call, the module's static storage pointer is automatically saved in the Dispatch Table. When the kernel calls the new system call handler routine, it passes the saved memory address in the **a3** register, permitting the system call handler to access the static storage using the symbolic names with which the static storage was defined. This effectively allows the System Globals memory structure to be extended as needed.

11.4 USER AND SYSTEM STATE CALLS

There are two Dispatch Tables – a user table and a system table. When a system call is made the kernel uses one or the other table to fetch the appropriate routine address (indexed by the system call code). The choice is made on the basis of the processor state of the caller, determined by inspecting the saved status register on the stack. Each table is 512 long words. The first group of 256 entries are the addresses of the system call handler functions, indexed by the system call code. The second group of 256 entries are the memory addresses appropriate to the system calls (set by the **FSSvc** call), also indexed by the system call code.

The **FSSvc** system call takes a flag with each call being installed indicating whether the routine address should be put in both dispatch tables, or in the System Dispatch Table only (for privileged calls). A function can be made to behave differently depending on whether the call was made from user or system state by installing the user state version of the routine in both tables, and then the system state version of the routine in the System table only.

This is commonly done by the kernel for I/O system calls. A user state call has its path number translated through the path number conversion table in its process descriptor, to give the system path number identifying the appropriate path descriptor. A system state call passes the system path number directly.

11.5 THE SYSTEM CALLS

Each system call provided by OS-9 version 2.4 is listed below, with its function code and a brief description of its purpose. Many of the function codes are historical, and are no longer used. Others have been defined for future use, or for special applications. These unimplemented codes are shown with the description in *italics*. Privileged system calls are shown with a **!**

symbol preceding the description. System calls that only apply if the System Security Module is in use are shown with "SSM" preceding the description.

The I/O system calls are described in detail in the section on the I/O System.

Some system calls were created by Microware specifically for use in their utilities, and are not documented in the OS-9 Technical Manual, while other system calls were not documented until recently. These calls are shown with a ■ after the description, and are briefly described in the sections following the table below.

<u>Code</u>	<u>Name</u>	<u>Description</u>
\$000	F\$Link	Link to a module in memory.
\$001	F\$Load	Read a file of one or more modules into memory and install the modules as a group in the module directory. ■
\$002	F\$UnLink	Unlink a module, specifying the module address. From OS-9 version 2.3 onwards, if the call is from user state and the SSM is in use, the module header must be in the caller's memory map. If the unlink reduces the link count to zero, (or -1 for a sticky module), the kernel deletes the module from the module directory, subject to certain checks that the module is not in use. A module whose type code is greater than 12 is assumed to be an I/O module, and the kernel makes an F\$IODEl system call to check that it is not in use by an I/O sub-system. From OS-9 version 2.4.3 (released in 1992), the kernel checks each process descriptor to ensure that the module is not a primary program module or an installed trap handler.
\$003	F\$Fork	Start a process, specifying a program module to link to or file name to load. The kernel first attempts to link to a module of the given name. If that fails, the kernel attempts to load from a file of the given name, relative to the execution directory. If this succeeds, the kernel will execute the module loaded (irrespective of the name). If the file contains more than one module, the first module is executed. The link and load requests are executed on behalf of the new process, not the parent process.
\$004	F\$Wait	Wait for any child of this process to die.
\$005	F\$Chain	Convert this process to executing a new program module.
\$006	F\$Exit	Terminate this process.
\$007	F\$Mem	Change the size of the primary data area (initial static storage and stack) of this process (not recommended for new applications). This call will fail if it attempts to expand the static storage, and insufficient contiguous free memory is available above the current static storage. The F\$Fork system call uses this system call to allocate the primary data area for the process. Memory is allocated from low memory upwards (contrast F\$SRqMem).
\$008	F\$Send	Send a signal to a process.

OS-9 SYSTEM CALLS

\$009	F\$Icpt	Install or replace the signal handler function for this process. A handler address of zero cancels any currently installed signal handler for the process.
\$00A	F\$Sleep	Put this process to sleep for a time, or until woken by a signal. A request for a sleep of one tick immediately re-inserts the process in the active queue, causing a reschedule. A request for a sleep of n ticks will sleep until n-1 tick interrupts are received.
\$00B	F\$SSpd	<i>(Suspend Process).</i>
\$00C	F\$ID	Return the process ID of this process.
\$00D	F\$SPrior	Set the execution priority of a process.
\$00E	F\$STrap	Install a handler function for "hardware" exceptions (bus error, address error, illegal instruction, and so on).
\$00F	F\$PErr	Print an error number, with an optional description string searched for in a text file.
\$010	F\$PrsNam	Parse the name of a module or file.
\$011	F\$CmpNam	Compare a match string with a file or module name, including wild card characters.
\$012	F\$SchBit	Search a bit map for a free (clear) field.
\$013	F\$AllBit	Allocate (set) a field in a bit map.
\$014	F\$DelBit	De-allocate (clear) a field in a bit map.
\$015	F\$Time	Get the current data and time.
\$016	F\$STime	Set a new date and time, or read the date and time from a battery-backed clock (by specifying a date of zero).
\$017	F\$CRC	Generate or check the module CRC over part or all of a module (or other memory area).
\$018	F\$GPrDsc	Get a copy of the process descriptor of a process.
\$019	F\$GBlkMp	Get information about the free memory list. ■
\$01A	F\$GModDr	Get a copy of the module directory.
\$01B	F\$CpyMem	Copy from an absolute memory address to the caller's buffer (the process ID parameter mentioned in the OS-9 Technical Manual is not used).
\$01C	F\$SUser	Change the user and group numbers of this process. Only permitted in two cases. Firstly, if the caller is a super user (group 0). Or secondly, if the owner of the primary module (in the M\$Owner field of the program module header) is a super user and was at the time of forking (the kernel compares the M\$Owner field with the P\$MOwn field of the process descriptor), and the new user and group are to be the same as the module owner. Note: prior to OS-9 version 2.3 the check that M\$Owner had not changed was omitted.
\$01D	F\$UnLoad	Unlink a module by name.

\$01E	F\$RTE	Exit a signal handler function (this call must be used to ensure that all pending signals are processed).
\$01F	F\$GPrDBT	Get a copy of the process descriptor table.
\$020	F\$Julian	Convert a date and time in Gregorian format to Julian format.
\$021	F\$TLink	Link to a trap handler module and install it to handle future trap #n instructions from this process.
\$022	F\$DFork	Fork a process to be debugged.
\$023	F\$DExec	Execute one or more instructions of a child process being debugged.
\$024	F\$DExit	Terminate a child process being debugged.
\$025	F\$DatMod	Create a data module (or other module type) in memory. ■
\$026	F\$SetCRC	Correct the header parity and CRC of a module in memory.
\$027	F\$SetSys	Read or write a field of the System Globals.
\$028	F\$SRqMem	Allocate memory by priority only (no regard for colour). This call will not allocate memory of priority zero.
\$029	F\$SrtMem	De-allocate memory (return it to the free pool).
\$02A	F\$IRQ	⌘ Install an interrupt handler function in the interrupt polling table.
\$02B	F\$IOQu	⌘ I/O queue this process on another process (that is using an I/O resource). The queue is ordered by the scheduling constants of the processes at the time they were placed in the queue. A process being added to the queue is placed later in the queue than other processes in the queue with equal or greater scheduling constants.
\$02C	F\$AProc	⌘ Put a process in the active queue.
\$02D	F\$NProc	⌘ Make the first process in the active queue the current process.
\$02E	F\$VModul	⌘ Validate a module in memory and install it in the module directory.
\$02F	F\$FindPD	⌘ Get the address of a path or process descriptor, given the path number or process ID and the base address of the path or process descriptor table.
\$030	F\$AllPD	⌘ Allocate a new path or process descriptor, given the base address of the path or process descriptor table. Searches the table for a free entry (which determines the new path number or process ID), allocates the required memory and clears it, and sets the address in the table. Then sets the first word of the allocated memory to the path number or process ID, and returns the path number or process ID and the address of the allocated memory. The first two words of the table give information about the memory structures. The first word is the current maximum path number or process ID permitted (equal to the size of the table in long words, minus one). The second word is the size of each structure (path or process descriptor).

OS-9 SYSTEM CALLS

\$031	F\$RetPD	⌘ De-allocate a path or process descriptor, given a path number or process ID in d0 and the base address of the path or process descriptor table in a0 . Clears the table entry for this descriptor, and de-allocates the memory.
\$032	F\$SSvc	⌘ Install one or more system call handler routines.
\$033	F\$IODEl	⌘ Check unlinking of an I/O module (file manager, device driver or device descriptor). ■
\$037	F\$GProcP	⌘ Get the address of a process descriptor given a process ID. ■
\$038	F\$Move	⌘ Optimized memory copy (also takes into account any MMU restrictions).
\$039	F\$AllRAM	<i>(Allocate RAM blocks).</i>
\$03A	F\$Permit	SSM - Add memory area to process's memory map (permits the process to access the memory area). ■
\$03B	F\$Protect	SSM - Remove memory area from process's memory map. ■
\$03C	F\$SetImg	<i>(Set Process DAT Image).</i>
\$03D	F\$FreeLB	<i>(Get Free Low Block)</i>
\$03E	F\$FreeHB	<i>(Get Free High Block)</i>
\$03F	F\$AllTsk	⌘ SSM - Ensure the MMU is set up for this process. If the SSM is not installed, this call is not privileged - it does nothing, and returns no error. ■
\$040	F\$DeiTsk	⌘ SSM - De-allocate the task number for this process. If the SSM is not installed, this call is not privileged - it does nothing, and returns no error. ■
\$041	F\$SetTsk	<i>(Set Process Task DAT registers).</i>
\$042	F\$ResTsk	<i>(Reserve Task number).</i>
\$043	F\$ReiTsk	<i>(Release Task number).</i>
\$044	F\$DATLog	<i>(Convert DAT Block/Offset to Logical).</i>
\$045	F\$DATTmp	<i>(Make temporary DAT image).</i>
\$046	F\$LDAXY	<i>(Load A [X,[Y]]).</i>
\$047	F\$LDAXYP	<i>(Load A [X+,[Y]]).</i>
\$048	F\$LDDXY	<i>(Load D [D+X,[Y]]).</i>
\$049	F\$LDABX	<i>(Load A from 0,X in task B).</i>
\$04A	F\$STABX	<i>(Store A at 0,X in task B).</i>
\$04B	F\$AllPrc	⌘ Allocate a new process descriptor - calls F\$AllPD , then sets the current date and time in the P\$DatBeg and P\$TimBeg fields of the process descriptor.
\$04C	F\$DeIPrc	⌘ De-allocate a process descriptor, given the process ID in the d0 register. Calls F\$RetPD .
\$04D	F\$ELink	<i>(Link using Module Directory Entry).</i>

\$04E	F\$FModul	⌘ Find a module directory entry. ⌘
\$04F	F\$MapBlk	<i>(Map Specific Block).</i>
\$050	F\$ClrBlk	<i>(Clear Specific Block).</i>
\$051	F\$DeIRAM	<i>(De-allocate RAM blocks).</i>
\$052	F\$SysDbg	Invoke system level debugger. ⌘
\$053	F\$Event	Create, link to, unlink from, delete, change, inspect, or wait for an OS-9 event.
\$054	F\$Gregor	Convert a date and time in Julian format to Gregorian format. ⌘
\$055	F\$SysID	Get the system identification information. ⌘
\$056	F\$Alarm	Set up to be sent a signal after a timed interval, or periodically. In system state, install a handler function to be called after a timed interval, or periodically.
\$057	F\$SigMask	Increment, decrement, or clear the signal mask for this process. The d0 register must be zero. The d1 register must be -1 (to decrement the signal mask), or 0 (to clear the signal mask), or 1 (to increment the signal mask).
\$058	F\$ChkMem	SSM - check that a memory area is within this process's memory map. If "write" permission is requested, the SSM checks that the memory is not write protected from this process. Otherwise, it checks that the process can read and execute the memory. In user state, or if SSM is not used, this call just reads the first byte of the memory area (generating a bus error if the memory is not accessible). ⌘
\$059	F\$UAcct	For a user accounting module. The kernel makes this call when a process is forked, chained, or terminated. A kernel customization module can install a handler for this call, and maintain user accounting information.
\$05A	F\$Cctl	Enable, disable, or flush the processor program and/or data caches.
\$05B	F\$GSPUMP	SSM - get a copy of the memory map of a process. ⌘
\$05C	F\$SRqCMem	Allocate memory of a particular colour. This call will allocate memory of priority zero if no other memory of that colour is available.
\$05D	F\$POSK	<i>(Execute service request).</i>
\$05E	F\$Panic	Panic warning. The kernel has no handler for this call, but makes this call if all processes have been terminated. Custom modules could make this call under other fatal conditions, such as power failure. A watchdog module could install a handler for this call to handle these situations gracefully. Normally this call would be installed for system state use only.
\$05F	F\$MBuf	<i>(Memory buffer manager). This system call is implemented as part of the Internet Support Package (ISP).</i>

OS-9 SYSTEM CALLS

\$060	F\$Trans	Translate a memory address as seen by the CPU into the address to be used by an alternate bus master, using the address translation offset given in the memory list in the init configuration module.
\$080	I\$Attach	Ensure an I/O device is initialized.
\$081	I\$Detach	Terminate usage of an I/O device.
\$082	I\$Dup	Get another local path number for an open path.
\$083	I\$Create	Open a path and create a file.
\$084	I\$Open	Open a path to an existing file or device.
\$085	I\$MakDir	Create a directory file.
\$086	I\$ChgDir	Change the current data and/or execution directory for this process.
\$087	I\$Delete	Delete a file.
\$088	I\$Seek	Change the current file pointer on a path.
\$089	I\$Read	Read from a path without data editing.
\$08A	I\$Write	Write to a path without data editing.
\$08B	I\$ReadLn	Read from a path, terminating on [CR], allowing data editing.
\$08C	I\$WritLn	Write to a path, terminating on [CR], allowing data editing.
\$08D	I\$GetStt	Get information about a path, file, or device.
\$08E	I\$SetStt	Modify information or operation, or request special action, of a path, file, or device.
\$08F	I\$Close	Close a path.
\$092	I\$SGetSt	Get a copy of the device name or path descriptor options section of an open path using a system path number.

11.5.1 F\$AltTsk System Call

The kernel makes this system call just before starting or restarting a process in user state. It is a request to the SSM to ensure the MMU is correctly set up for the current process. Some MMUs can store multiple process memory maps simultaneously. The current map is selected by writing a number to the MMU. Under OS-9 this number is known as a task number.

If the MMU can store multiple maps the SSM first checks to see whether the map for this process is already in the MMU – that is, a task number is allocated to the process. In this case the SSM need only write the task number to the appropriate MMU register to select the map for the current process. The SSM stores the process's task number in the **P\$Task** field of the process descriptor. (If the process's map is not currently in the MMU the SSM sets **P\$Task** to some invalid value as an indication of this.) If no task

number is currently allocated to the current process the SSM tries to find an unallocated task number for it. Otherwise it must take a task number from another process.

In the case that the MMU cannot store multiple maps the SSM will keep a record of the process descriptor address of the process whose map is currently in the MMU, so that it does not unnecessarily rewrite the map in the MMU.

The SSM will therefore have decided whether the memory map for the current process must be written to the MMU. Some MMUs can read the processor's memory, so that they read the map themselves as necessary. For these MMUs (such as the MMUs in the 68030 and 68040) the SSM only needs to write the root address of the process's memory map to the MMU register. Otherwise the SSM must copy the process's memory map to the MMU internal memory.

The SSM will also need to copy the process's memory map to the MMU internal memory if the process's memory map has changed - the map previously stored in the MMU is "stale". The SSM **F\$Protect** and **F\$Permit** system calls set bit 4 of the **P\$State** field in the process's process descriptor to indicate that the memory map of the process has been changed. If the MMU uses internal memory to store the map (rather than directly accessing the processor's memory), the **F\$AllTsk** system call must update the map stored in the MMU if this bit is set, even if the map had been previously written to the MMU. The SSM then clears the bit flag, to indicate the map in the MMU is now up to date.

11.5.2 F\$CCtl System Call

The higher members of the 68000 family have on-chip memory caches. The 68020 has an instruction cache, while the 68030 and 68040 have separate instruction and data caches. In addition, some processor boards have off-chip caches. In order to be able to support such boards, Microware has not included control of the caches in the kernel. Instead, the caches are controlled by the **syscache** kernel customization module, which installs the **F\$CCtl** system call (the kernel's default handler for this system call does nothing, and returns no error). The system call allows the instruction and data caches to be separately enabled, disabled, and flushed (any dirty data is written to main memory, and the current cache contents are forgotten). The kernel uses this system call, for example to disable the data caches during I/O calls.

The parameter passed to the system call is not an image of the processor's **cacr** (cache control) register. Instead, it is a long word of six bit flags, each requesting the instruction or data caches to be enabled, disabled, or flushed. If an undefined bit is set, or a call is made from user state requesting action other than flushing one or both cache sets and the caller is not a super user, a "parameter" error (**E\$Param**) is returned. It is not an error to request an action that is not supported by the hardware (for example, enabling the data cache on a 68020).

This system call supports nested requests to disable the instruction or data caches, which are enabled on coldstart. If a request is made to disable the instruction or data caches, the **D_DisInst** or **D_DisData** field respectively of the System Globals is incremented, and the appropriate caches are disabled. If a request is made to enable a set of caches, the appropriate field of the System Globals is decremented (unless it is already zero). If it is now zero, the corresponding caches are enabled, otherwise they are left disabled (and the flag bit in the parameter is cleared). If the parameter contains flags requesting that a cache set be both enabled and disabled, the request to enable the cache takes precedence, and the request to disable the cache is ignored. If the parameter has no bits set this is taken to be a request to flush all the caches.

Note that the current state of the cache control is not maintained separately for each process. Therefore if a process disables caching it does so for all processes. The most recent flag settings are saved in the **D_CachMode** field of the System Globals. The parameter passed to the system call is:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	1	Pattern of bit flags.

The bit flags are defined as assembly language symbols in the file 'DEFS/process.a'. They are:

<u>Bit Number</u>	<u>Name</u>	<u>Description</u>
0	b_endata	Enable the data cache(s).
1	b_disdata	Disable the data cache(s).
2	b_fldata	Flush the data cache(s).
4	b_eninst	Enable the instruction cache(s).
5	b_disinst	Disable the instruction cache(s).
6	b_flinst	Flush the instruction cache(s).

The example below shows an assembly language function to make this system call, and a C call to it requesting that the data caches be disabled:

```
void dis_data_cache()          /* disable the data cache(s) */
{
    cache_ctl(0x02);           /* set bit 1 to disable data cache(s)
*/
}

#asm
cache_ctl:  os9      F$Cctl      the parameter is already in d0.1
            rts
#endasm
```

11.5.3 F\$ChkMem System Call

This system call checks whether a process has permission to access a memory area, by searching the SSM memory map of the process. The parameters to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	Size of the memory area.
d1	b	Access permissions requested - read, write, execute (same format as disk file modes byte).
a2	l	Address of the memory area.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist.

The kernel uses this system call whenever a system call made from user state passes a pointer to a memory area for the system call to read or write - for example, an I/O "read" (**I\$Read**) request.

11.5.4 F\$DatMod System Call

This system call creates a module in memory. Prior to OS-9 version 2.3 only a data module could be created, and the use of coloured memory was not possible. From OS-9 version 2.3 onwards an extension to the system call allows the module type and language to be specified explicitly, and a memory colour to be specified. These two extra parameters are used only if bit 15 of the **d2** register (module permissions) is set, otherwise the system call assumes a type of "data", a language code of zero, and a colour of zero (general system memory). The parameters passed to the call are:

OS-9 SYSTEM CALLS

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	Size of the body of the module desired (excluding the header, CRC, and name string).
d1	w	Module attributes and revision number.
d2	w	Module access permissions. Also, if bit 15 is clear, registers d3 and d4 are ignored.
d3	w	Module type and language.
d4	b	Memory colour to use (zero means general system memory).
a0	l	Address of the name of the module to create.

The kernel allocates memory for the module, builds the module header, clears out the module body, sets the module CRC, and installs the module in the module directory. The "execution offset" field (**M\$Exec**) of the module header gives the offset from the start of the module header to the module body – the memory for use by the program. The values returned are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	Module type and language.
d1	w	Module attributes and revision number.
a0	l	Caller's register updated past name string.
a1	l	Address of module body ("execution entry").
a2	l	Address of module header.

11.5.5 F\$DelTsk System Call

The kernel makes this system call when terminating a process. It indicates to the SSM that the task number (if any) which is allocated to the process can be released for use by another process. Or, if the MMU can only store one map, that the SSM should forget that the MMU contains the map for this process (in case the process descriptor memory is reused for another process). The SSM also de-allocates any remaining memory that it had allocated for the management of this process's memory map.

11.5.6 F\$FModul System Call

The **F\$Link** system call uses this function to locate a module in the module directory, given the module name, type, and language. The parameters to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	Module type (high byte) and language (low byte) (or zero to ignore type or language).
a0	l	Address of the module name string.

This system call searches the module directory for the desired modules, and returns:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	Actual module type and language.
d1	w	Module attributes (high byte) and revision number (low byte).
a0	l	Updated past the module name.
a2	l	Address of the module directory entry.

11.5.7 F\$GBlkMp System Call

This system call returns information about the free memory areas on the system. The parameters passed to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	Memory areas whose start address is below this value are not included in the returned segment list (but are included in the segment count and free memory total).
d1	l	Size of the caller's buffer to contain the returned segment list (in bytes).
a0	l	Address of the caller's buffer.

The system call routine scans the free memory lists, counting the number of separate segments of memory that are free in the system, and totalling their size. For each such segment whose start address is above the specified minimum passed in the **d0** register, the start address and size are written to the buffer (as long words, in that order), until the buffer is exhausted. If the buffer is not exhausted when all segments have been scanned, the next entry in the buffer is cleared to zeros (two long words).

If a segment lies in a memory area that is not designated as "user" memory in the coloured memory list in the **init** configuration module, the segment is not included in the totals or in the table. The values returned from the call are:

OS-9 SYSTEM CALLS

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	System minimum allocatable block size, copied from the D_BlkJiz field of the System Globals.
d1	l	Number of separate free segments of memory found.
d2	l	Total amount of RAM found at startup, copied from the D_TotRAM field of the Systems Globals.
d3	l	Total of free user memory at present.

11.5.8 F\$GProcP System Call

This system call returns the address of a process descriptor, given the process ID. The parameters passed to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	The process ID.

The values returned from the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
a1	l	Address of the process descriptor.

11.5.9 F\$Gregor System Call

This is the complement to the **F\$Julian** system call. It converts a Julian date and time to Gregorian format. The parameters to the system call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	Julian time.
d1	l	Julian date.

The system call returns:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	"Gregorian" time.
d1	l	Gregorian date.

A Julian date is simply the number of days from a reference date. A Julian date of zero corresponds to a Gregorian date of 2nd January, in the year -4712. The Gregorian date of the 1st January, 1900, corresponds to a Julian date of 2415020. A Julian time is simply the number of seconds since midnight. While Gregorian dates and times are more natural to humans, the

Julian equivalents are easier to manipulate numerically. Therefore OS-9 provides system calls to translate between the two representations.

Under OS-9 Gregorian dates are held within a single long word, with bits 16:31 containing the year (since 0 AD – for example, 1992), bits 8:15 containing the month, and bits 0:7 containing the day of the month. This is often represented as **YYYYMMDD**. Similarly, "Gregorian" times are held in a single long word, with bits 16:23 containing the hour (24 hour clock), bits 8:15 containing the minute, and bits 0:7 containing the second. This is often represented as **00HHMMSS**.

11.5.10 F\$GSPUMp System Call

This system call returns a generalized representation of the memory map of a process. The SSM converts its memory map for the process into a standard table form in the caller's buffer. The table is an array of word values, one for each memory block in the address space of the processor, where the block size is the System Minimum Allocatable Block Size (**D_BlkSiz** in the System Globals) – the block size supported by the MMU. A typical block size is 4k bytes. For a processor with a 32 bit address bus (as the 68000 family has) – a 4 Gigabyte address space – this would give a table 2M bytes in size!

To avoid the need for such a large buffer, Microware have taken into account that in reality most systems have all their user memory low down in the address space. The System Globals field **D_AddrLim** contains the address of the highest memory location (both RAM and ROM). The caller can use this field, and the **D_BlkSiz** field, to determine the size of table to allocate. In any case, the SSM will not return information for address space blocks above this limit.

Many systems have the RAM low down in the address space, and the ROM high up in the address space. This can result in a very large desired table size. The calling program may decide that it does not need the mapping information for the ROM areas. The memory list in the **init** module can be searched to determine the actual extent of the RAM space on the system.

The table is therefore a representation of the address space of the processor, or a part of it starting at address zero. For example, if the system has 4M bytes of RAM starting at address zero, and the System Minimum Allocatable Block Size is 4k bytes, the table requires 1024 word entries (2048 bytes). For each block of 4 kbytes, starting at address zero, the SSM will build a table entry with the following format:

High byte:	Access permissions	
	<u>Bit</u>	<u>Permission if set</u>
	0	Read
	1	Write
Low byte:	2	Execute
	Use count.	

If the block is not in the memory map of the designated process, the entry is set to zero. The SSM also returns the System Minimum Allocatable Block Size (a copy of the value in **D_BlkSiz**). The parameters to the system call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	Process ID of the process whose memory map is desired.
d2	l	Size of the buffer for the table (in bytes).
a0	l	Address of the buffer for the table.

The system call returns the following values:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	System Minimum Allocatable Block Size.
d2	l	Size of the table (in bytes) – equal to the buffer size, unless limited by D_AddrLim .

11.5.11 F\$IODEl System Call

This system call checks that a module is not in use by any device table entry. The kernel makes this call internally whenever a file manager, device driver, or device descriptor module is unlinked, reducing the link count to zero. If the module is in use by a device table entry, the kernel leaves the module's link count at one, and returns an error number 209 (**E\$ModBsy**). The parameters to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
a0	l	Address of the module to check.

11.5.12 F\$Load System Call

This system call reads a file containing one or modules, allocating memory for the modules, and installing them as a module group in the module

directory. If the "read" bit is set in the file access modes parameter, and the "execute" bit is not set, the file is opened relative to the process's current data directory. Otherwise it is opened relative to the process's current execution directory. Prior to OS-9 version 2.3 it was not possible to specify the colour of the memory to load the modules into. From OS-9 version 2.3 onwards an additional parameter giving an explicit memory colour is taken if bit 7 of the **d0** register (the file access modes) is set, otherwise general system memory is used. The parameters passed to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	b	File access modes. Also, if bit 7 is clear, register d1 is ignored.
d1	b	Memory colour to use (zero means general system memory).
a0	l	Address of pathlist of file to load from.

The values returned are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	w	Module type and language.
d1	w	Module attributes and revision number.
a0	l	Caller's register updated past pathlist string.
a1	l	Address of program start ("execution entry").
a2	l	Address of module header.

If the file contains more than one module, the returned values are for the first module in the file. The link count of the first (or only) module is set to one. The link counts of any other modules in the file are set to zero. See the chapter on OS-9 Memory, Modules, and Processes for a description of module groups.

11.5.13 F\$Permit System Call

If the SSM is in use, a process cannot normally access memory other than memory allocated to it, or modules it has linked to. An attempted illegal access will fail (a "write" will not affect the destination memory), and a bus error exception will be generated. However, in some applications it is necessary for a program to access other areas of memory. This system call allows a process to gain access to any memory area. The **F\$Permit** system call is also used internally by the kernel whenever it wishes to add a memory area to a process's memory map – for example, when a process allocates memory, or links to a module.

OS-9 SYSTEM CALLS

As with other SSM system calls, the functionality of this call depends on the implementation within the SSM. The description here is of the functionality of a Microware SSM for the memory management unit in the 68030 processor. This system call adds a memory area to the memory map of a process, permitting the process to access that memory. The parameters to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	l	Size of the memory area.
d1	b	Access mode requested (read, write, execute) in the same format as path modes.
a2	l	Address of the memory area.

If the call is made from user state, an error is returned unless the call is made by a member of the super user group (group zero). If the process was forked by the **F\$DFork** system call, as a debugged process, the memory area is also added to the parent's memory map. This is recursive, so a debugged process can fork a debugged process, and so on.

The memory address is rounded down to the nearest whole block of the size supported by the MMU (the System Minimum Allocatable Block Size). The size is rounded up to a whole number of blocks, to include the start and end addresses of the area requested. This ensures that the whole of the desired memory area is accessible to the process, without the process needing to know the system minimum allocatable block size.

If the process is the System Process (the process ID is one), the system call does nothing.

Read and execute permissions are always granted (the 68030 MMU does not distinguish between them). Write permission is only enabled if requested in the access modes.

The SSM must take account of the possibility that **F\$Permit** will be called more than once for the same process and memory area. For example, a process might link to the same module several times. Therefore the SSM searches the process's memory map to see if the block is already in the process's memory map. If so, the SSM just increments a use counter that it keeps for each block in a process's memory map. Otherwise it adds the block to the process's memory map, and sets the use counter for that block to the initial value of one.

This feature is necessary so that when a request is made to unmap the block – for example, when a module is unlinked – it is only actually removed from the process's map when the block is no longer required for any reason. (See the description of **F\$Protect** below). Note that the precision of the use count may be limited – for example, the SSMs for the 68030 and 68040 maintain an 8 bit use counter for each block. Therefore a possibility of error exists if a block is "mapped to" more than 255 times by the same process (for example, if a module is linked to more than 255 times by the same process). The SSM limits the counter to 255, so after 255 "unlinks" the module will be removed from the map of the process, even though the process believes it is still linked to the module.

If the MMU has internal memory maps, the SSM does not update the MMU during this system call. Rather, it updates the map in memory that will be copied to the MMU when the process is next about to execute in user state (see the description of the **F\$AllTsk** system call above). While in system state the MMU is configured to remove all protections – operating system functions (and system state programs and trap handlers) have unrestricted access to the full memory map of the processor.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist, causing the calling process to be aborted (unless it has installed a bus error handler – see the **F\$STrap** system call, and the chapter on Exception Handling).

11.5.14 F\$Protect System Call

This system call is the complement to **F\$Permit**. It removes a memory area from the memory map of a process, denying the process access to that memory. The parameters to the call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	1	Size of the memory area.
a2	1	Address of the memory area.

If the call is made from user state, an error is returned unless the call is made by a member of the super user group (group zero). If the process was forked by the **F\$DFork** system call, as a debugged process, the memory area is also removed from the parent's memory map. This is recursive, complementing the nested **F\$DFork** calls supported by **F\$Permit**.

As with **F\$Permit**, the memory address is rounded down to the nearest whole block of the size supported by the MMU (the System Minimum Allocatable Block Size). The size is rounded up to a whole number of blocks, to include the start and end addresses of the area requested.

If the process is the System Process (the process ID is one), the system call does nothing.

The system call decrements the use count in the process's memory map for each block in the memory area (see the description of **F\$Permit** above). If the use count for a block reaches zero, it is removed from the process's memory map.

If the SSM is not used (and so has not installed a handler for this system call), the kernel's default handler simply reads the first byte of the memory area. This will generate a bus error if the memory is not accessible, or does not exist.

The kernel uses this system call whenever a memory area is de-allocated by a process, or the process unlinks from a module.

11.5.15 F\$SysDbg System Call

This system call causes the kernel to call the "system debugger", if one is present. The kernel makes a subroutine call to the routine whose address is in the **D_SysDbg** field of the System Globals. The kernel only checks that the caller is the super-super user (user zero of group zero). It does not set up any registers for the "system debugger" (although the **a6** register contains the address of the System Globals). From OS-9 version 2.3 onwards, the kernel makes the **F\$CCtl** system call to flush and disable the data and instruction caches before calling the debugger, and to flush and enable the caches on return from the debugger.

The **D_SysDbg** field of the System Globals is initialized during the kernel's coldstart to the "boot entry point" address passed from the boot program, plus 16. In the boot program, this should point to a branch instruction to the ROM-based debugger. Therefore executing the **F\$SysDbg** system call normally invokes the ROM-based debugger. This halts the normal operation of the system. The ROM-based debugger will continue normal system operation in response to the "go" (g[CR]) command.

11.5.16 F\$SysID System Call

The kernel header contains a licensee number, a serial number, the processor type this kernel supports, and (in an encrypted form prior to OS-9 version 2.3) a version description string and a copyright string. The body of the kernel also contains an author names string in an encrypted form. This system call returns these items (with the strings decrypted as necessary), together with the processor type in use (determined dynamically by the boot program).

The processor type numbers are the Motorola part number: 68000, 68010, 68020, 68030, 68040, 68070 and so on. For example, a 68010 processor running the 68000 version of the OS-9 kernel would give a kernel processor type of 68000, and the processor type in use as 68010.

The strings returned are null terminated, and will not be longer than 80 characters, including the null. The parameters to the system call are:

<u>Register</u>	<u>Size</u>	<u>Description</u>
a0	1	Address of the buffer for the version string.
a1	1	Address of the buffer for the copyright string.
a2	1	Address of the buffer for the author names string.

If a buffer address is passed as zero, the system call handler does not attempt to copy that string. Apart from copying the strings to the buffers, the system call returns:

<u>Register</u>	<u>Size</u>	<u>Description</u>
d0	1	OS-9 licensee number.
d1	1	Serial number of this copy of OS-9.
d2	1	Processor type in use.
d3	1	Kernel processor type.
d4-d7	1	Zero.

The licensee number and serial number are one by default, but may be used by Microware or the licensee (the manufacturer of the computer system) to identify the licensee and the individual copy of OS-9, as a piracy protection measure.

CHAPTER 12

DEVICE DRIVERS



This section is intended to dispel the mystery surrounding device drivers. It explains the purpose of a device driver within OS-9, describes the operating system environment the device driver works in, and shows typical algorithms for particular device types. Particular attention is given to the use of interrupts, as interrupt-driven devices are central to the proper functioning of a multi-tasking computer (under any operating system).

The descriptions and code fragments in this section assume that the device driver is written in 68000 assembly language. However, device drivers can equally well be written in C. The section on Microware C and Assembly Language describes how this is done.

12.1 THE FUNCTION OF A DEVICE DRIVER

A device driver is one part of an OS-9 I/O sub-system. All I/O system calls go initially to the kernel. However, the kernel has no understanding of the filing structure of a device, or of the hardware used to control the device. The job of the kernel is simply to set up the software environment, such as allocating a path descriptor, or locating an existing path descriptor if the path is already open. The kernel must then call the file manager or the device driver to carry out operations on the device. Similarly, the file manager understands the data structure of the device, but it does not know how to handle the hardware. If the file manager wishes to perform device operations – such as a data transfer – it must call the device driver.

In summary, the OS-9 I/O sub-system philosophy is to split the I/O operations as follows:

DEVICE DRIVERS

- The kernel allocates and de-allocates path descriptors, device table entries, and device static storages.
- The file manager handles the filing structure and any data editing.
- The device driver carries out physical device operations.

The device driver is therefore generally only concerned with performing low level physical device operations, without any understanding of why these are be carried out. However, this is not a strict requirement of OS-9. Device drivers may perform data interpretation, or other higher-level functions. For example, serial port drivers generally recognize certain special characters, such as the abort and interrupt keys. This is necessary because these keys must be acted upon as soon as they are received – that is, within the interrupt service routine. The file manager would only "see" the keys when a process subsequently performs a read request, which may be much later (or never!).

Similarly, a serial port driver for communications work might also incorporate a communications protocol because a received packet must be acknowledged within a very short time of reception, or because the file manager being used does not understand the protocol. It is sufficient that the combination of the file manager and the device driver provides all the data manipulation and hardware control functions.

The I/O system is normally a simple tree structure. There is one kernel, which can call multiple file managers. Each file manager can call multiple device drivers. Although it is conceivable that different file managers could call the same driver, in practice this is rarely done, because each file manager will normally have a different calling convention, and different structures for the path descriptor and device static storage. The **RBF** and **PCF** file managers are an example of this technique. **PCF** carefully uses the same path descriptor and device static storage structures as have been defined by the writer of **RBF**, so that **PCF** can implement an MS-DOS filing system using existing **RBF** device drivers.

12.2 DEVICE STATIC STORAGE

When considering the operation of an I/O sub-system within a multi-tasking operating system it is important to distinguish between "logical" paths and "physical" devices. A path is an operating system construct to enable a program to make system calls for data transfer and control on a device. The operating system can create more paths (given enough memory), up to some large limit (65535 under OS-9). A device is (normally) a construct to allow

the operating system to control a physical hardware object, such as a disk drive or a serial port. The operating system can "create" new devices, but usually only on a one-to-one correspondence with the hardware objects. Multiple paths may be open on a single device, but a path cannot be open on multiple devices. Note that this abstracted description relates to the kernel's view of devices. At a lower level (for example, in a device driver), the definition may become blurred. For example, a pipe does not relate to any hardware object – it uses a memory buffer only. Also, a device driver could use multiple I/O chips to provide one "device" with a complex function.

Because at any one time a device may have multiple paths open to it, or no paths, the path descriptor is not a suitable place to store variables for the control of the device. There must be one data structure in memory for each device. This is the purpose of the device static storage. Thus, the path descriptor is used to control the logical path, while the device static storage is used to control the physical device. The device driver is usually mainly concerned with the device static storage, as the device driver has the job of controlling the device. However, it may make use of some of the fields in the path descriptor, as these may provide information about the current configuration required for this particular I/O call.

Conversely, the file manager mainly makes use of the path descriptor for variables storage, as the file manager has the job of managing the logical path – for example, input line editing for a "read line" system call. But it may make use of some device static storage fields. For example, RBF maintains some disk structure information in the device static storage, as it is needed by all path-related functions using the device.

The kernel is responsible for allocating and de-allocating the device static storage for an I/O sub-system. The kernel allocates a new device static storage when a new device descriptor is installed in the device table (by the **I\$Attach** system call, made implicitly when a path is opened) if either:

- The port address in the device descriptor is different from any other in the device table.

Or:

- Another device table entry has the same port address in its device descriptor, but a different device driver.

That is, the kernel considers this to be a new device if there is no device already in the device table using the same port address and device driver. Thus the kernel ensures that there is a separate device static storage allocated for each device currently in existence in the device table. Note that

by specifying a different device driver or by adjusting the port address in the device descriptor the programmer can coerce the kernel into allocating a separate device static storage for what may in fact be the same I/O interface. This is used in the SCSI driver system, where multiple drivers use the same I/O interface, and in the drivers for dual serial port chips, where separate "incarnations" of the driver must be created for each channel of the chip (because the SCF file manager does not support multi-channel drivers).

Conversely, the kernel permits multiple device table entries (for different device descriptors) that refer to the same device (they use the same device driver and port address). These "alias" device descriptors may be used to manage separate "channels" on the same device – such as multiple floppy disk drives attached to one controller – or to select different configurations for the device.

The device static storage comprises three or four parts. The first part is defined by the kernel – its size and usage is the same for all devices. Note that some fields are not used by the kernel – they have been defined for the convenience of the file manager and device driver writers, because they are required by many file managers and device drivers. This is an example of how Microware has provided a comfortable environment for the device driver writer, to try and limit the extent to which the programmer must learn about the operating system before writing a device driver. (Nonetheless, it is strongly recommended that you learn as much as possible about the operating system before writing a device driver.)

Following the kernel section, the file manager may (and usually does) require its own storage. The size and usage depends on the file manager, and is defined by the file manager writer. This part will be the same for all devices controlled through the same file manager. If the file manager supports multi-channel devices (for example, four disk drives attached to one interface) it will also usually require an area of storage for each channel, known as a drive or channel table. Therefore the third part of the device static storage – which exists only if the file manager supports multi-channel devices – contains the drive tables. This is simply an array of structures (drive tables), one for each channel, usually indexed by a logical drive or channel number (base zero). Typically the file manager will be capable of supporting a large number of channels, but on any given device the actual number will be far fewer. Therefore to conserve memory the size of the array is determined by the device driver (which knows how many channels the hardware will support) rather than the file manager.

The last part (highest memory address) of the device static storage is the storage area required by the device driver. Its size and usage is determined by the device driver writer. It effectively comprises the static variables of the device driver. It is defined using the normal statements for static variable definition in the source files of the device driver, whereas the kernel and file manager parts are defined in separate source files, as described below. The total size of the device static storage for the kernel to allocate is taken from the memory size entry in the device driver module header (**M\$Mem**). It is the sum of the kernel, file manager, and device driver requirements. The size is in the device driver module header because the device driver is at the bottom of the tree, so the total size is only known when the device driver is created. It is calculated by the linker when creating the device driver module.

To simplify this operation, Microware provides pre-prepared definitions of the kernel and file manager storage, already assembled, to give the static storage (**vsect**) definitions required to reserve this storage. They need only be included at link time when creating the device driver. For example, when linking an SCF device driver:

```
$ 168 ../LIB/scfstat.1 RELS/sc6850.r -l=../LIB/sys.1
-O=OBSJ/sc6850
```

and when linking an RBF device driver for a device supporting two drives:

```
$ 168 ../LIB/drvs2.1 RELS/rb1772.r -l=../LIB/sys.1
-O=OBSJ/rb1772
```

Note that 'drvs2.1' is simply a merging of three ROFs:

```
$ chd /dd/LIB; merge rbfstat.r drvstat.r drvstat.r
>drvs2.1
```

'drvs2.1' reserves storage for the kernel and RBF, and for two RBF drive tables (as this device driver supports two drives on one interface).

The file 'LIB/scfstat.1' is an assembly of the file 'DEFS/scfstat.a', and is used for device drivers that work with the SCF file manager. The file 'LIB/rbfstat.r' is an assembly of the file 'DEFS/rbfstat.a', and is used for device drivers that work with the RBF file manager. It defines the storage required by the kernel and RBF. An RBF device driver must also reserve device static storage for the RBF drive tables, using the file 'LIB/drvstat.r' (which is an assembly of the file 'DEFS/drvstat.a') once for each drive to be supported. The files 'LIB/drvs1.1', 'LIB/drvs2.1', and 'LIB/drvs4.1' contain 'LIB/rbfstat.r' followed by one, two, or four copies (respectively) of 'LIB/drvstat.r', for device drivers supporting one, two, or four drives. By merging 'LIB/rbfstat.r' and multiple copies of 'LIB/drvstat.r' you can create versions for any number of disk drives.

Note that the copy of 'LIB/rbfstat.r' must come first in the "merge", as the definitions for the kernel and file manager storage must precede the drive tables. Similarly, in the linker command line the appropriate static storage definition file (such as 'LIB/scfstat.l') must precede the driver ROF (or ROFs), as all of the kernel and file manager storage must precede the driver storage in the device static storage memory. The linker builds the final static storage definitions from the static storage definitions in the ROFs strictly in the order that the ROFs appear on the linker command line. This ensures that kernel references into the device static storage are correct, even though the kernel does not know the structure (or even the existence) of the file manager and driver parts of the device static storage. Similarly, file manager references are correct, even though the file manager does not know the structure of the driver part.

As described above, the source files for these storage definitions are in the 'DEFS' directory (typically '/dd/DEFS'), as is a "make" file ('DEFS/makefile') to assemble and merge them. The equivalent files for the SBF file manager are 'LIB/sbfstat.r' (for the kernel and file manager definitions), and 'LIB/sbfdrvtrb.r' (for one drive table), although full source code is not provided with OS-9. The SBF static storage structures are fully defined in 'DEFS/sbfdev.d', and in the files listed in Appendix B.

After allocating a new device static storage the kernel clears it to zeros. This is a very convenient software flag mechanism. If the initialization routine of the device driver aborts due to an error, the termination routine is always called by the kernel. The termination routine can "clear up" what resource allocation or device initialization the initialization routine managed to do before aborting, by looking to see if initialization flags are non-zero. For example, a field might be used to store the address of an allocated memory buffer, and will only be non-zero if the memory was actually allocated. However, the kernel does not support full C-like static storage initialization of the device static storage.

The device driver is not restricted to using only the device static storage for its variables and buffers. It can allocate additional memory as required using any of the available memory allocation mechanisms, including the general memory allocation system call (**F\$SRqMem**), coloured memory, and data modules. Unlike the management of a program's memory allocations, the kernel does not keep track of memory allocated by a device driver (or any operating system component). It is the responsibility of the termination routine of the device driver to ensure that all such memory is de-allocated. The same philosophy applies to other resources that the device driver may allocate, such as creating an event or opening a path.

12.3 PATH DESCRIPTOR

A path descriptor is a memory structure (always 256 bytes) used by OS-9 to manage a path. Because the path concept is concerned with the logical manipulation of data (data editing, filing structure, and so on), the path descriptor variables in the first 128 bytes are mainly used by the kernel and the file manager, not by the device driver. The second half of the path descriptor is the "options section". It contains a copy of the options section of the device descriptor on which the path was opened. The layout of the options section is defined by the file manager writer, although the device driver writer may define additional fields in the device descriptor (but outside of the options section proper), pointed to by the offset value in the **M\$DevCon** field of the device descriptor extended module header.

The options section contains parameters used to select optional behaviour of the file manager and device driver. For example, an SCF options section contains all the line editing key codes, and other special characters, while an RBF options section contains disk format parameters. The file manager may also dynamically write additional fields at the end of the path descriptor options section that are not defined in the device descriptor options section. These fields are for the information of a program, which can read all 128 bytes of the path descriptor options section using a Get Status request with the function code **SS_Opt**. For example, RBF puts a copy of the file name in the **PD_NAME** field of the options section.

The options sections of the Microware file managers are described in the OS-9 Technical Manual. The following paragraphs do not repeat those descriptions. Instead, they attempt to clarify certain areas that have caused difficulty to users in the past.

12.3.1 RBF Path Descriptor

RBF is not concerned with the physical layout of the disk (cylinders, surfaces, physical sector numbering¹⁴). It uses a logical sector numbering convention in which sector 0 is the first sector on the disk, and the other sectors are numbered sequentially. Some controllers (such as SCSI controllers) use the same convention, so the device driver does not need to translate. Otherwise,

¹⁴ Disk drive terminology is often confused. A disk drive will have one or more disks on the same spindle. Each disk has one or two data surfaces. Data is read and written in concentric rings on each surface, using a read-write head on each surface. Each ring on each surface is a track, while the rings on all surfaces at the same radius are known as a cylinder. Therefore the total number of tracks equals the number of cylinders multiplied by the number of surfaces. Within each track the data is subdivided into equal sectors, numbered from zero or one upwards on each track.

DEVICE DRIVERS

the device driver must use the fields in the options section of the path descriptor to convert the logical sector number (LSN) to a cylinder number, surface (or head) number, and sector number. The calculation can be somewhat complex. The following fields of the options section are relevant:

PD_DRV	Logical drive number. This is the number used (base zero) by RBF to index into the drive tables. It may also be used by the device driver as the physical drive number, if the drives are numbered sequentially from zero upwards (but see PD_LUN).
PD_CYL	Number of cylinders available for data (for LSN validity check, and partitioning).
PD_SID	Number of data surfaces (tracks per cylinder).
PD_SCT	Number of sectors on each track, except track zero (cylinder zero, surface zero).
PD_TOS	Number of sectors on track zero.
PD_TOffs	First physical cylinder (<i>not track</i>) to use. After calculating the cylinder number (base zero) from the LSN, this value must be added to the cylinder number to form the true physical cylinder to access. This feature is used to skip cylinder zero on the Microware Universal floppy disk format, as different controllers place different restrictions on the format used on track zero.
PD_SOffs	First sector number on each track (zero or one). After calculating the sector number (base zero) within the track from the LSN, this value must be added to the sector number to form the true sector number to access. This feature is used because certain disk formats number the sectors on a track from zero, while others number the sectors from one.
PD_LUN	Physical drive number. If the interface is connected to multiple controllers (as with SCSI), then this is the drive number on that controller. This field may be equal to PD_DRV if only one controller is supported by the interface, and the drives are numbered sequentially from zero. This is typically the case for a simple floppy disk controller.
PD_LSNOffs	Offset for logical sector numbers. The driver must add this value to the LSN supplied by RBF before using the LSN in a controller command, or converting it to physical parameters. This allows support for partitioning on a hard disk.
PD_TotCyls	Total physical cylinders on the disk (for formatting, and LSN validity check). This value is usually equal to PD_CYL plus PD_TOffs (or the sum of these for all partitions) plus any allowance for cylinders reserved for automatic defect handling by the disk controller.
PD_CtrlrID	Controller number. This field is only used if the interface can be connected to multiple controllers (as with SCSI).

In addition to the parameters described above, other format variables are specified in the path descriptor options section:

PD_TYP

Disk type flags. If bit 7 is set, the disk is a hard disk, otherwise it is a floppy disk. This bit is only of importance for controllers that support both hard and floppy disk drives. Prior to OS-9 version 2.4, bit 0 was set for an 8" disk, and reset for a 5¼" (or 3½") disk. An 8" disk requires a rotational rate of 360rpm, and a data rate (MFM) of 500kbps, while a 5¼" disk requires a rotational rate of 300rpm, and a data rate (MFM) of 250kbps. Note that as far as the floppy disk controller and device driver are concerned, there is no difference between a 5¼" disk and a 3½" disk. As more disk formats were developed this became restrictive, and from OS-9 version 2.4 onwards bit zero is not used. Bits 1:4 define the disk size:

Value	Disk size
1	8" disk
2	5¼" disk
3	3½" disk

while the rotational speed and data rate are defined in the new field **PD_Rate**. If bits 1 to 4 of **PD_TYP** are zero, the driver knows that the descriptor is from before OS-9 version 2.4, and so bit zero of **PD_TYP** is used, and **PD_Rate** is not defined.

Two further bits are defined. If bit 5 is set, track zero (cylinder zero, surface zero) is double density, otherwise it is single density. This allows the support of old formats that have single density (FM) on track zero, and double density (MFM) on other tracks. Finally, for a hard disk (bit 7 is set), if bit 6 is set, the hard disk is removable.

PD_DNS

Data density flags - if bit 0 is set, the disk is double density (MFM encoding), otherwise it is single density (FM encoding). Single density is used only rarely today, in support of old formats on products using historical standards.

PD_Rate

This field is defined for OS-9 version 2.4 onwards. Bits 0:3 specify the rotational speed:

Value	Speed
0	300rpm (3½" or 5¼")
1	360rpm (8", or PC-AT 5¼")
2	600rpm

and bits 4:7 specify the data rate:

Value	Data rate
0	125kbps (3½" or 5¼" single density only)
1	250kbps (3½" or 5¼" double density, or 8" single density)
2	300kbps (ditto, but rotating at 360rpm)
3	500kbps (high density, or 8" double density)
4	1000kbps
5	2000kbps
6	5000kbps

Combinations of these fields allow the support of all commonly used floppy disk formats. However, not all controllers and disk drives will support all rotational speeds and data rates.

PD_SSize

Number of bytes per sector. This is the block size RBF will assume when requesting data transfers. Prior to OS-9 version 2.4 only a value of 256 was permitted. From OS-9 version 2.4 onwards any value that is a power of 2, from 256 to 32768, is permitted. Also, when opening a file, and before

DEVICE DRIVERS

performing any data transfers or allocating any data buffers, RBF will make a Get Status call to the driver with the function code **SS_VarSect**. This gives the driver the opportunity to check or alter the value in **PD_SSize** to correspond to the medium in use. For example, a device descriptor for a SCSI hard disk drive may have zero in this field. The device driver updates the field with the actual disk sector size returned from the drive controller in response to the SCSI READ CAPACITY command.

If the driver returns no error in response to the **SS_VarSect** call, RBF assumes the value in **PD_SSize** is the correct sector size to use. If the driver returns the error **E\$UnkSvc** (unknown request), RBF assumes a default sector size of 256 bytes. (Any other error code will cause RBF to abort the opening of the file with an error.) Having determined the sector size, RBF writes it to the path descriptor options section field **PD_SectSiz** (a long word). It is this value that RBF uses for subsequent operations, and it can be read by a program, using the Get Status request **SS_Opt** to return a copy of the options section.

Microware supports a range of floppy disk formats. Although the preferred distribution format is the Universal format (which does not use cylinder zero), this is a recent standard, and many OS-9 systems use other - older, or more conventional, or higher data density - formats. The user may therefore have a number of different "alias" device descriptors for the same floppy disk drive, specifying different format parameters. The Microware-defined format codes depend on the density of track zero - single density (FM) or double density (MFM) - and the number of the first sector on each track (sector offset). In addition, the Universal format (code 38U0) does not make use of cylinder zero - the cylinder offset is one. The commonly used formats are:

Format code	Track 0 density	Sector offset	Cylinder offset
3803	FM	0	0
3807	MFM	0	0
38W7	MFM	1	0
38U0	MFM	1	1

The format codes shown above are for 3½" disks. The initial "3" of the format code is changed to "5" for 5¼" disks. All of these formats use double density on all tracks (other than track zero for 3803 format), 80 cylinders, and 16 sectors per track (10 sectors on track zero for 3803 format).

A device driver written to support a wide range of formats will need to take account of all of the above parameters when initializing the disk controller, and when performing data transfers. There is the risk for a removable disk

(such as a floppy disk) that the user will insert a disk of a different format. In this case the user will normally access the disk using a different device descriptor, with the appropriate format parameters. The device driver must check at each transfer (or perhaps only when a path is opened – RBF makes the Set Status call **SS_Open** to the driver) whether the format parameters have changed, requiring a re-initialization of the disk controller.

One simple way of doing this is to keep a record (in the device static storage) of the address of the device descriptor last used to initialize the controller. The address of the device descriptor can be taken from the device table entry. The address of the device table entry for this device is in the path descriptor location **PD_DEV** (it is set up by the kernel), and the field **V\$DESC** in the device table entry contains the address of the device descriptor. If the current device descriptor address is different from the one last used to initialize the controller, a re-initialization is required. This is not foolproof – the user might change the parameters in an already loaded device descriptor, or unlink a device descriptor and load a new one at the same address – but it is simple and effective. The alternative is to check each one of the format parameters against the values last used to initialize the controller.

Certain fields of the options section are used to control the behaviour of RBF and the device driver:

PD_VFY	Disable verify after write. If this field is non-zero the device driver should not perform a verify (read) of each sector after writing it. Verify-after-write is only used with disk structures that do not support error detection and correction – usually floppy disks. Other device drivers (for example, for SCSI hard disk drives) will ignore this field.
PD_SAS	Minimum segment allocation size. When RBF is asked to extend a file (for example, by a write at the end of the file), if the extension is shorter than this value (in sectors) RBF will allocate this many sectors to the file. When the file is closed, RBF trims back the file to its true length (provided the file pointer is at the end of the file). This reduces the fragmentation problem caused by two files "leap-frogging" each other as they are written to.
PD_Cnt1	A word of bit flags, having the following effects when set: 0 – enable formatting and writing to sector zero. If this bit is not set, the driver should return an error ES\$Format if it is requested to format the disk, or to write to sector zero. Hard disk device descriptors are usually format protected in this way, with a special device descriptor being loaded in order to format the disk, or set a new boot file (the os9gen utility writes the address of the boot file to sector zero). 1 – enable multi-sector transfers. If this bit is not set, RBF will only request the device driver to transfer one sector with each request. This bit should be reset for controllers that can only transfer one sector at a time.

DEVICE DRIVERS

3 - the device driver can determine the disk capacity. If this bit is set, the device driver supports the **SS_DSize** Get Status call, returning the disk capacity in sectors.

PD_MaxCnt Maximum transfer size. RBF will not ask the device driver to transfer more than this number of bytes in one request. This may be set to a limit imposed by a DMA controller, for example. If the controller cannot transfer more than a certain number of sectors in one request, either this field must be set to that number of sectors multiplied by the sector size, or the device driver must be able to divide up a large request into manageable pieces.

The RBF path descriptor variables section (the first 128 bytes of the path descriptor) contains three fields of interest to the device driver writer:

PD_DEV Address of the device table entry for the device on which the path was opened. The device driver can use this pointer to get the address of the device descriptor (field **V\$DESC** in the device table entry).

PD_DTB Address of the drive table. RBF multiplies the logical drive number (**PD_DRV**) by the size of one drive table, and adds it to the address of the first drive table in the device static storage, to form this address. The device driver must copy the first 22 bytes of LSN zero to the drive table at this address whenever LSN zero is read or written.

PD_BUF Buffer address. When RBF calls the read or write routines of the device driver, this field contains the address of the memory to read to or write from.

12.3.2 SCF Path Descriptor

The SCF path descriptor options section is mainly composed of special key codes for line editing and keyboard signals. If a key code is set to zero, an incoming character is not checked against the key code. This permits line editing functions to be disabled. Note that the **xmode** utility allows the user to modify the options section of an SCF device descriptor in memory, while the **tmode** utility modifies the path descriptor options section for path 0, 1, or 2 (standard input, standard output, and standard error) of its inherited paths. Certain other options fields modify the line editing behaviour of SCF. Of these, the most commonly used are:

PD_EKO Enable echo. If this field is non-zero, SCF echoes each character as it is read during a "read" (**I\$Read**) or "read line" (**I\$ReadLn**) request.

PD_ALF Automatic line feed. If this field is non-zero, SCF outputs a line feed character (\$0A) after each carriage return character (\$0D) in a "write line" (**I\$WritLn**) request.

PD_PAU End of page pause. SCF counts carriage return characters (\$0D) as they are written by "write" and "write line" requests. It resets the count if any "read" or "read line" request is made. If this field is not zero, when the count

reaches the value in **PD_PAG** SCF does not output the carriage return character until a character has been received. Note that this applies even to characters written by "write" requests, so it is important to clear this field when sending binary data.

PD_EOR	End of record character (usually [CR]). If this field is non-zero, SCF compares every incoming character with this field, and terminates a "read" or "read line" request when a character matches this field. Note that if echo is enabled (see PD_EKO) the carriage return character (\$0D) is echoed in response to this character being received.
PD_EOF	End of file character (usually [ESC]). If this field is non-zero, and a matching character is read as the first character of a "read" or "read line" request, SCF aborts the request with an "end of file" error (E\$EOF). For a "read line" request, the end of file condition is reported if the first character in the edit buffer matches this field, even if other characters have previously been entered and then erased.
PD_PSC	End of line pause key code (usually [^W]). SCF copies this field to the V_PCHR field of the device static storage. The "data received" interrupt service routine of the device driver should compare each incoming character with the V_PCHR field (if non-zero). If a match is found, the device driver sets the V_PAUS field of the device static storage. When SCF is about to write a carriage return character from a "write" or "write line" request, it checks the V_PAUS field. If non-zero, SCF waits for a character to be received (other than a match for PD_PSC) before outputting the carriage return.
PD_INT	Interrupt key code (usually [^C]). SCF copies this field to the V_INTR field of the device static storage. The device driver "data received" interrupt service routine should check each incoming character against this field (if non-zero). If a match is found, the driver sends an interrupt signal (3 - S\$Intrpt) to the last process to use the device. The process ID of this last process is copied by the kernel to the field V_LPRC in the device static storage. (V_LPRC is set to zero by SCF if the process has died, and it has no parent. If it has a parent with a path open on the same device, the parent's ID is copied to V_LPRC). When SCF sees this character as part of a "read line" request, it acts as if the "delete line" key code had been received.
PD_QUIT	Abort (or quit) key code (usually [^E]). Similar to PD_INT , SCF copies this field to the V_QUIT field of the device static storage, and the device driver sends an abort signal (2 - S\$Abort) if a matching character is received.

Two fields - **PD_XON** and **PD_XOFF** - are used for software flow control. They usually have values \$11 and \$13 respectively ([^Q] and [^S]) - the ASCII XON and XOFF characters. SCF copies these fields to the **V_XON** and **V_XOFF** fields of the device static storage, and the device driver uses these values (if non-zero) as the character codes to restart and stop transmission (respectively) in both directions. That is, if an XOFF character is received, the driver suspends transmission until an XON character is received. Conversely, if the driver's receive buffer is becoming full (it has reached a "high water mark"), it sends an XOFF character, and then sends an XON

DEVICE DRIVERS

character when the buffer has emptied by some pre-determined amount (it is reduced to a "low water mark").

Lastly, the device driver uses two fields to determine the desired configuration of a serial port (if that is what is being controlled):

PD_PAR Character format. The bits of this field are used as follows:

0:1 parity generated and expected:

0 none

1 even

3 odd

2:3 bits per character:

0 8

1 7

2 6

3 5

4:5 number of stop bits:

0 1

1 1.5

2 2

Not all devices will be able to support all character formats.

Typically a device driver will configure the device to automatically pause transmission if the CTS handshake input is negated (relying on the fact that in most circuit designs this input will float asserted if not connected), and will assert the RTS (or DTR) handshake output. The driver will also configure the device to generate an interrupt when the DCD handshake input changes state (relying on the fact that in most circuit designs this input will float securely to either the asserted or negated condition if not connected). However, some device drivers (not Microware's) also use bits 6 and 7 to control the use of the hardware handshake lines:

6 set to disable hardware handshake (RTS/CTS)

7 set to disable recognition of DCD input

PD_BAU Baud rate code – see the table of baud rates below.

<u>Code</u>	<u>Rate</u>	<u>Code</u>	<u>Rate</u>
0	50	9	2000
1	75	10	2400
2	110	11	3600
3	134.5	12	4800
4	150	13	7200
5	300	14	9600
6	600	15	19200
7	1200	16	38400
8	1800	255	external

An "external" baud rate is set in hardware, and is not controllable by the device driver. Not all devices will be able to support all baud rates. Note that

no means is given of separately defining the baud rate for receive and transmit.

The device driver for a serial port will use these values to configure the interface during the driver's initialization routine. However, a program may wish to dynamically change the configuration on an open path. The program can modify these fields in the path descriptor options section using the Set Status system call with the **SS Opt** function code (the **_ss_opt()** C library function). SCF will copy the new values to the path descriptor options section, and then pass the call on to the driver. On receiving this call the driver should check whether the device configuration fields in the path descriptor have changed since the last time the interface was initialized. If so, the driver should re-initialize the interface.

If the device driver is controlling an "intelligent" communications board, the board may also support the detection of the flow control and signal characters. For this type of device, the driver should also check to see whether these fields have changed. If any of the configuration fields have changed, the driver should re-initialize the board.

While this dynamic re-configuration capability of the device driver is desirable, early Microware example SCF device drivers did not provide this feature. As a result there are many SCF device drivers in existence that take no notice of changes to the device configuration fields of the path descriptor.

If a device descriptor or path descriptor options section specifies a device configuration that the device (or the device driver) does not support, the device driver should return a "bad mode" error (**E\$BMode**).

12.4 SYMBOLIC DEFINITIONS

Microware have provided symbolic definitions in both C and assembly language for the structures and constants likely to be used by a device driver. These files are all in the 'DEFS' directory – it is strongly recommended that you study all of these files carefully before writing a device driver. The assembly language files are pre-assembled to make the library 'LIB/sys.l'.

Therefore a device driver written in assembly language does not need to pull in (**use** assembler directive) any of these files – the references are resolved at link time. Device drivers written in C will need to **#include** the relevant files. A typical list for an RBF device driver might be:

DEVICE DRIVERS

<code>rbf.h</code>	Path descriptor options section structure.
<code>MACHINE/reg.h</code>	Processor register definitions.
<code>procid.h</code>	Process descriptor structure.
<code>path.h</code>	Path descriptor variables section format.
<code>module.h</code>	Module header structures (including device descriptor).
<code>errno.h</code>	Error codes.
<code>signal.h</code>	Signal codes.
<code>sg_codes.h</code>	Set Status and Get Status function codes.

Note that the order of the **#include** statements for these files is important, as some files declare structures that are used in other files. The only operating system structure for which there is not a proper C definitions file is the System Globals. The file 'DEFS/setsys.h' does give the offsets within the System Globals structure to each field, but the System Globals is not defined as a structure, and the fields are not "typed".

In this book the symbolic names for the OS-9 error codes are sometimes given using the assembly language definitions in the file 'DEFS/funcs.a', and sometimes given using the C language definitions in the file 'DEFS/errno.h'. The symbol names are the same in both sets of definitions, except that the C definitions start with `E_` and use upper case only, while the assembly language definitions start with `E$` and use both upper and lower case. For example, the C symbol for the "not ready" error (code 246) is `E_NOTRDY`, while the assembly language symbol is `E$NotRdy`.

12.5 REGISTER USAGE

OS-9 was originally written completely in assembly language, although parts are now written in C. Therefore parameters are passed to and returned from the device driver in processor registers. In the following descriptions, as elsewhere in this book, parentheses around a register name mean "points to", and a suffix of ".b", ".w", or ".l" gives the size of the object in the register as byte, word (16 bits), or long (32 bits). Where the object is smaller than the register containing it, the object is always in the low order bits of the register, starting with bit zero.

Because the initialization and termination routines of the device driver are called directly by the kernel (as part of the **I\$Attach** and **I\$Detach** system calls), the calling convention to these routines is defined by the kernel, and is therefore the same for all drivers:

- (a1) Device Descriptor module
- (a2) Device Static Storage
- (a4) Process Descriptor of calling process
- (a6) System Globals

The calling conventions for the other routines (usually read, write, get status, and set status) are determined by the file manager, and may vary, especially for the read and write routines. Usually, however, the following conventions are adhered to:

- (a1) Path Descriptor
- (a2) Device Static Storage
- (a4) Process Descriptor of calling process
- (a5) Caller's register stack frame
- (a6) System Globals

The other registers may contain other parameters. The return convention for read and write varies according to the file manager. The error return convention for all of the functions is (usually) the same as that used throughout the operating system: the carry flag of the Condition Codes register is set if there was an error, in which case **d1.w** contains the appropriate OS-9 error code.

The kernel saves all the registers it uses before calling the initialization and termination routines of the device driver, except for the **a6** register when calling the termination routine. Therefore the driver need only preserve the stack pointer and the high byte of the status register (and **a6** in the termination routine). In general, file managers also save all the processor registers they use before calling the device driver functions. This is true of SCF and RBF. However, other file managers may only save certain registers, in order to speed up calls to the device driver, so it is important to check the documentation on the file manager.

12.6 DEVICE DRIVER ROUTINES

A device driver is a separate OS-9 module, so the addresses of its routines are not known to the kernel and file manager. However, the kernel and file manager need to be able to call the device driver routines. To achieve this, the "execution entry offset" (**M\$Exec**) in the module header of the device driver gives an offset from the start of the module to a table of offsets from the start of the module to each of the routines.

DEVICE DRIVERS

Two routines are absolutely required – initialization and termination – as these are called by the kernel when an I/O sub-system is created and deleted.

Any other routines are only called by the file manager, and their presence or absence is a matter for the file manager specification. File managers can specify any number of device driver routines for any purpose. Conventionally, however, the file manager requires four routines, making six in total. The code fragment below shows a typical **psect** statement and routine offset table for a device driver:

```

                use      /dd/DEFS/oskdefs.d
Typ_Lang        equ      (Drvrr<<8)+0bjct    module type and language
Att_Revs        equ      ((ReEnt+SupStat)<<8)+0  attributes and revision
*              number
Edition         equ      1                    software edition number
psect           sc68681,Typ_Lang,Att_Revs,Edition,0,EntryTable

EntryTable      dc.w      Init                initialize
                dc.w      Read                input data
                dc.w      Write               output data
                dc.w      GetStat             wildcard call (I$GetStt)
                dc.w      SetStat             wildcard call (I$SetStt)
                dc.w      Term                terminate
                dc.w      0                   exception handler (see below)
```

Notice that the last parameter to the **psect** statement is the label of the routine offset table. It is from this statement that the linker takes the value to put in the "execution entry offset" field of the module header. Note also that the positions of the initialize and terminate routine offsets within the table are fixed, as these routines are called by the kernel. Therefore if the file manager does not require one or more of the read, write, get status, or set status routines these entries must still exist (replacing the routine label with zero), and if the file manager needs additional routines their offsets must be added to the end of the table shown above.

Microware have indicated that future versions of the kernel may implement an additional "exception handler" routine, using a seventh table entry as shown above. This entry point will be called if a hardware exception (such as bus error) occurs during driver execution.

12.6.1 Initialize

This routine is called directly by the kernel. It is only called when the I/O sub-system is being created – that is, a new device static storage has been allocated. It is *not* called on the first usage of each channel on a

multi-channel device. The kernel calls the initialization routine as part of the **I\$Attach** system call. This call can either be made explicitly by a program (such as the **iniz** utility), or implicitly whenever a path is opened on the device. The initialization routine may be called more than once as the I/O sub-system is terminated and then re-created, but there will always be an intervening call to the termination routine as part of the termination of the I/O sub-system.

Some I/O devices must only be initialized once after reset – a repetition of the reset would cause problems. There is no operating system mechanism to determine whether this is the first time this I/O sub-system has been brought into being since reset. If it is important to know this, the initialization routine can use a data module. It attempts to create a data module whose name is constructed from the device port address. If there is no error, the data module did not already exist (otherwise a "known module" error **E_KWNMOD** would be returned), so this is the first time the I/O sub-system is being created.

This data module mechanism is also useful for sharing hardware with one or more other drivers. Common variables (such as the current state of write-only registers) can be held in the data module. Also, if the module does not already exist on initialization the driver knows it is the first user and must initialize the hardware. If the initialization and termination routines maintain a use count in the data module, the termination routine can know that it is the last user, and must terminate the hardware.

The initialization routine has a number of responsibilities. It must:

- a) Initialize the device static storage as needed. Usually the driver only initializes its own fields of the device static storage, but it may initialize other fields to pass device information to the file manager. For example, an RBF driver sets the field **V_NDRV** to the number of drives supported (which must be no larger than the number of structures in the drive table), and the **DD_TOT** field of each drive table structure to a non-zero value (to permit RBF to read LSN zero).
- b) Initialize the hardware, ready for subsequent calls to the other routines, such as read and write. A device driver expecting unrequested data to be received (such as an asynchronous serial port) must also set up the device ready for data to be received. For example, the interface chip would

be set up to generate interrupts when characters are received.

- c) Install the interrupt service routine in the polling table (**F\$IRQ** system call) if the device is to be interrupt driven. If the device driver is to receive interrupts on more than one vector, it will need to install multiple interrupt service routines. OS-9 places no limit on the number of interrupt service routines one driver can install.

The initialization routine is passed the address of the device descriptor module, not the address of the path descriptor (there may be no open path if an explicit **I\$Attach** system call is being made). However, the Microware definitions in the 'LIB/sys.l' library only include definitions for the offsets into the options section of the path descriptor – there are no symbolic definitions for accessing the options section of the device descriptor. As the two options sections have (by definition) the same structure, the programmer can use the same symbols – with a constant offset – to access the options section of the device descriptor. For example:

```
move.b PD_BAU+M$DTyp-PD_OPT(a1),d0
```

will access the baud rate code in the device descriptor (assuming the **a1** register is pointing to the device descriptor), while:

```
move.b PD_BAU(a1),d0
```

will access the baud rate code in the path descriptor (assuming the **a1** register is pointing to the path descriptor). This works because **M\$DTyp** is the offset from the start of the device descriptor to the first entry in the options section, while **PD_OPT** is the offset from the start of the path descriptor to the start of the options section.

The device driver should not sleep as part of the initialization routine. The kernel does not build the device table entry until the initialization routine returns, so a concurrent I/O call from another process on the same device would cause a recursive call to the **I\$Attach** system call and the initialization routine of the device driver. Also note that the **V_BUSY** field of the device static storage is not set to the process ID of the calling process at this time (see below), as this is a function of the file manager.

12.6.2 Terminate

The termination routine is essentially the converse of the initialization routine. It is called directly by the kernel as part of the dismantling of an I/O sub-system, from within the **I\$Detach** system call. The kernel will only call the termination routine when the device use count (in the device table entry)

has been decremented to zero. That is, there are no paths open on the device, and any explicit calls to **I\$Attach** (and **I\$ChgDir**) have been complemented by an equal number of explicit calls to **I\$Detach**. The kernel will always de-allocate the device static storage after calling the termination routine, and (in all versions of the kernel to date) ignores any error returned by the termination routine.

The termination routine must:

- a) Wait for any "write-behind" activity to finish. For example, characters may be waiting in a buffer to be transmitted out of a serial port under interrupt, perhaps paused by software or hardware handshake.
- b) Shut down the hardware. In particular, the hardware must be disabled from generating any interrupts or other autonomous behaviour.
- c) De-allocate any resources allocated by the device driver. Examples are buffer memory allocated, data modules created or linked to, paths opened, and events created or linked to.
- d) Remove the driver from the interrupt polling table, using the **F\$IRQ** system call. Each interrupt service routine that the driver installed must be un-installed.

Note that if the initialization routine returns an error to the kernel, the **I\$Attach** system call will call the termination routine before de-allocating the device static storage.

12.6.3 Read

As mentioned above, the read routine (if it exists) is only called by the file manager. Therefore the purpose of the routine and the parameter convention used when calling it are determined by the file manager writer. In general it is used to get data from the device. As an illustration, for this and the other routines the purpose and parameter convention are shown for the SCF and RBF file managers. These two file managers adhere to the general parameter convention described above, so only the additional parameters are described below.

□ The Sequential Character File Manager (SCF)

Purpose: read one character.

Parameter convention:

Passed: nothing

Returns: **d0.b** = character read

SCF drivers usually maintain a circular input buffer in the device static storage (or dynamically allocated in the initialization routine) filled under interrupt. The interrupt service routine for the "data received" interrupt takes the character from the chip and puts it in the circular buffer. The driver read routine takes a character from this buffer, waiting (by sleeping) if the buffer is empty. The interrupt service routine is responsible for waking up the driver when a character is received, and for detecting and acting on certain special characters – flow control (XON and XOFF), "interrupt" (usually [^C]), "quit" (usually [^E]), and "end-of-line pause" (usually [^W]).

The "data received" interrupt service routine is also responsible for sending the "pause" flow control character (XOFF) when the buffer is becoming full – usually at a "high water mark" of three quarters full. Conversely, the read routine is responsible for sending the "restart" flow control character (XON) if a "pause" had been requested and the buffer is now sufficiently empty – usually at a "low water mark" of one quarter full.

Characters may be received with errors. For example, parity checking may be enabled for an asynchronous serial port, and a character may arrive with incorrect parity. As this error status is normally supplied by the interface chip with each character, the "data received" interrupt service routine must save the error status as it reads each character from the chip. The standard Microware drivers simply bitwise OR the error status of each character into the **V_ERR** field of the device static storage. The interrupt service routine also sets a bit in this field if the input buffer overflows, so one or more characters are lost. The read routine checks this field when returning a character – if it is not zero, the routine clears the field and returns a "read" error (**E\$Read**).

Under this scheme the calling program is not able to determine which character was in error. This is not important when simply reading from a keyboard, but may be unsatisfactory for communications applications. A device driver for such an application might maintain a second circular buffer containing a status byte for each received character. When SCF requests a character for which the status is not zero, the driver returns a "read" error (**E\$Read**), and saves the status in the device static storage field **V_ERR**. Such a driver could also support the Get Status call function **SS_ELog** (read

error log), returning a copy of the latest saved error status, permitting the program to determine the type of error.

If no characters are available in the input buffer, the read routine must sleep. It is then woken by the interrupt service routine when a character arrives. This is described in detail below in the discussion of interrupts. The read routine may also be woken from its sleep by a signal from another process (sent to the process that called the driver), or by a "quit" or "interrupt" signal sent to the process by the interrupt service routine on receipt of one of the special key codes. The read routine must decide whether to go back to sleep and wait for a character, or to abort the read with an error. A typical device driver for terminals and printers will abort only if the signal received was a "deadly" signal. Prior to OS-9 version 2.4 the deadly signals were signal 0 (the kill signal - **S\$Kill**), signal 2 (the quit signal - **S\$Abort**), and signal 3 (the interrupt signal - **S\$Intrpt**). From OS-9 version 2.4 onwards all signals below 32 are considered deadly, except signal 1 (the wakeup signal - **S\$Wake**).

The use of an input buffer filled under interrupt provides a "type ahead" capability. That is, provided the device is active (a path is open to the device, or the device has been explicitly initialized), characters can be received in advance of any read request from a program. This allows a user to type in a command in advance of the previous command completing. More importantly, it reduces the real-time response requirement of a program that is receiving data. Typically an SCF device driver has an input buffer of 80 characters. Thus a program can delay 80 character times (about 80ms at 9600 baud) before reading the data without losing any data.

There is no requirement under OS-9 that device drivers must be interrupt driven. The read routine of an SCF device driver could simply poll the status register of the interface chip until a character had been received, and then return that character to SCF. However, this would destroy the multi-tasking capability of the operating system, as rescheduling does not take place while a system call is executing - the system call must go to sleep or exit to allow a reschedule.

An alternative approach is to poll the status register and, if no character is available, to sleep for one tick. Sleeping for one tick requests a reschedule, but the process remains active. This allows another active process (if there is one, and it is of sufficient priority - see the chapter on Multi-tasking) to become the current process, otherwise the driver (or rather, the process calling the driver) remains the current process, and continues to poll the status register.

DEVICE DRIVERS

Clearly, the most efficient technique – both in terms of processor time usage, and of speed of response to a received character – is to use interrupts. However, the above description shows that OS-9 does not force any particular style of operation on the device driver.

□ The Random Block File Manager (RBF)

Purpose: read one or more consecutive sectors from a disk.

Parameter convention:

Passed: **d0.l** = number of sectors

d2.l = starting Logical Sector Number (base 0)

PD BUF(a1) = memory to read to

Returns: nothing

Note: prior to OS-9 version 2.4 the number of sectors was in **d0.b** only, and could not exceed 255. Now the number of sectors is only limited by the **PD_MaxCnt** field of the path descriptor, which sets a limit on the total number of bytes RBF may ask the driver to transfer. The same change applies to the write routine.

RBF is not concerned with the physical disk structure. It uses a Logical Sector Numbering scheme (base zero). The device driver must (if necessary) convert this to physical disk parameters, as described above in the section on the Path Descriptor.

RBF is also not concerned with retries. If there is an error on reading, it is up to the driver to decide whether to try again to read the sector (or sectors). If the driver returns an error to RBF, then RBF will consider it to be an unrecoverable error, and abort the filing operation, which may cause some damage to the disk structure. The disk controller may do retries itself, in which case the device driver will not itself implement any retries. SCSI hard and floppy disk controllers typically operate in this way. For simple floppy disk controllers the device driver may retry several times, occasionally restoring (seeking to cylinder zero) and re-seeking, in case the problem is a head misalignment.

On reading (or writing) LSN zero the driver also has the responsibility to copy the first 22 bytes into the first part of the appropriate drive table structure in the device static storage. (Note that RBF has pre-calculated the address of the drive table structure for this drive, and placed it in the path descriptor field **PD_DTB**.) This 22 byte structure contains information about the disk structure, both logical and physical (refer to the OS-9 Technical Manual section on the RBF Drive Table). A device driver may elect to use the physical format fields of this structure in place of some of the fields of the

path descriptor when calculating cylinder, surface, and sector numbers. This allows a driver to dynamically adapt to different disk formats (although of course it does assume that the driver can read LSN zero). The useful fields are:

DD_TOT	Total data sectors on disk (maximum LSN plus 1) (for LSN validity checking). RBF will not issue a request for an LSN greater than or equal to this value. Therefore the initialization routine of the device driver must set this field non-zero in each drive table structure, to allow RBF to read LSN zero.
DD_TKS	Sectors per track. (The field DD_SPT contains the same value as a word rather than a byte).
DD_FMT	Disk format flags. The bits have the following meanings when set: <ul style="list-style-type: none"> 0 double sided disk 1 double density disk 2 double track density disk

For example, if bit 1 of the path descriptor field **PD_DNS** is set, indicating that the drive is double track density, but bit 1 of **DD_FMT** read from LSN zero is not set, indicating a single track density disk, then the driver knows that it must instruct the drive controller to "double step", that is, to move the head by two steps for each cylinder number (because the drive supports cylinders twice as closely packed radially as the disk has on it). Similarly, even though the drive supports double sided floppy disks (**PD_SID** in the path descriptor is 2), if bit 0 of **DD_FMT** is not set the driver knows that the disk is single sided, and so adjusts its calculation of the physical parameters from the LSN.

12.6.4 Write

The write routine of the device driver is usually very much the complement of the read routine, the only difference being the direction of data transfer – the write routine (generally) sends data to the device. Much of the code for the read and write routines is shared in most device drivers. As in the read routine, the specific function of the write routine is defined by the file manager specification. The basic requirements of an SCF device driver and an RBF device driver are shown below as examples.

□ The Sequential Character File Manager (SCF)

Purpose: write one character.

Parameter convention:

Passed: **d0.b** = character to write

Returns: nothing

SCF drivers usually maintain a circular output buffer in the device static storage (or dynamically allocated in the initialization routine) filled by the write routine, and emptied under interrupt. The interrupt service routine for the "transmitter ready" interrupt takes the character from the circular buffer and puts it in the chip. The driver write routine waits (by sleeping) if the buffer is full, and is woken by the interrupt service routine when the buffer has emptied a little.

Once the buffer has been completely emptied the interrupt service routine must disable further "transmitter ready" interrupts from the chip. Therefore the write routine must check whether the "transmitter ready" interrupts are disabled, and if so it must write the character directly to the chip (rather than putting it in the buffer), and enable the interrupts. This starts a stream of interrupts, each one being serviced by putting the next character into the chip. The stream is only stopped when the buffer becomes empty – the calling program has no more characters to send, or is supplying them at a rate below the transmission rate of the chip.

The "data received" interrupt service routine or the read routine may wish to send a flow control character (XON or XOFF), which must take precedence over any characters waiting in the output buffer. If "transmitter ready" interrupts are disabled, the flow control character is put directly in the chip, and the interrupts are enabled (starting a transmission stream that may be only one character long). Otherwise a flag is set in the device static storage. The flag is checked by the "transmitter ready" interrupt service routine at the next interrupt, and the flow control character is sent instead of taking the next character from the output buffer.

If the output buffer is full when SCF calls the write routine to send a character, the write routine must sleep. It is then woken by the interrupt service routine when space becomes available in the output buffer. Typically the interrupt service routine will not wake the write routine when just one space is available, but waits until the buffer has subsided to a low water mark, typically 10 characters left to send. This reduces the number of sleep/wakeup cycles, so reducing the processor load. This is described in detail below in the discussion of interrupts.

The use of an output buffer provides a "write behind" mode of operation. That is, provided there is sufficient space in the output buffer (typically 140 bytes in size), a program making a write request is returned to immediately, and continues with further operations while the data is transmitted at the rate permitted by the interface chip. This can be very important in preventing unacceptable delays – for example, when a real-time process prints an error or status message. However, under certain circumstances it may cause problems, as a program may need to know when a packet of data has completed transmission. In such applications the driver might be modified to provide an additional Set Status function to send a signal when the output buffer becomes empty.

The write routine may also be woken from its sleep by a signal from another process (sent to the process that called the driver), or by a "quit" or "interrupt" signal sent to the process by the "data received" interrupt service routine on receipt of one of the special key codes. The write routine must decide whether to go back to sleep and wait for a character, or to abort the write with an error. As with the read routine, a typical device driver for terminals and printers will abort if the signal received was a "deadly" signal.

SCF provides a read-write lockout. That is, even if the read routine goes to sleep (allowing another process to execute and make system calls), the write routine will not be called until the read routine has woken up and exited. A process making a system call that requires a write call to this device will be "I/O queued" until the read request finishes. The same mechanism applies if a read request is made while a write request is in progress. This mechanism is explained in detail in the section on Resource Control in the chapter on File Managers. It greatly simplifies the job of the device driver – which can use common device static storage locations for read and write calls – as well as ensuring that a message cannot appear on a display while a program is waiting for input (which would lose the waiting program's prompt).

□ The Random Block File Manager (RBF)

Purpose: write one or more consecutive sectors to a disk.

Parameter convention:

Passed: **d0.1** = number of sectors

d2.1 = starting Logical Sector Number (base 0)

PD BUF(a1) = memory to write from

Returns: nothing

This routine is very much the complement of the read routine, and most RBF device drivers will use the same subroutines for most of both operations. As with the read routine, the write routine must translate the RBF logical

parameters to the appropriate physical parameters for the disk controller, initialize the controller whenever the format parameters have changed, and copy the first 22 bytes of the data to the drive table whenever LSN zero is being written (provided the write operation is successful).

In addition to writing the data, a device driver for a controller that does not support error detection and correction (EDC) should verify that the write operation was successful, by reading the sector that has just been written. Some controllers implement a "verify" command that reads the sector to check it, but without returning the data to the interface. In the absence of such a facility the device driver must read the sector to a local buffer in the device static storage (or dynamically allocated in the initialization routine). The controller will generate an error condition if the sector was not written successfully. The device driver can then retry the write operation a few times, eventually returning a "write" error (**E\$Write**) to RBF if the sector cannot be written successfully.

For maximum confidence of data integrity the driver can compare the data read back by the verify operation with the data in the write buffer. This will reveal any errors in the transfer of data between the interface and the controller.

If the **PD_VFY** field of the path descriptor is not zero the driver should not perform the verify operation. Verifying after each sector is written is very time consuming, because the controller must wait for the disk to rotate a complete revolution before reading the sector that was just written. Therefore some programs – such as the **copy** and **backup** utilities – set the **PD_VFY** field non-zero while writing large blocks of data, to speed up data transfers.

RBF provides the same read-write lockout as described above for SCF. This greatly reduces the complexity of the device driver, and is appropriate because most block-structured devices cannot support concurrent read and write operations. The lockout is for the whole device. Therefore RBF will not call the driver to read or write on this or another drive on the same interface while a previous read or write request is not yet complete.

12.6.5 Get Status and Set Status

These are "wild card" routines. That is, they are a mechanism to permit any function to be implemented. By convention, the Get Status routine is used to request information from the device or driver, while the Set Status routine is used to request device or driver operations.

In general the file manager will perform the same device lockout for these routines as for the read and write routines, so it is permissible for the device driver to sleep as part of one of these calls. However, SCF does not implement device lockout in its Get Status routine. Therefore SCF device drivers must not sleep in a Get Status routine, or else the driver must implement the device lockout itself. In practice, if the SCF device driver writer wishes to add extra functionality to get information from the device, and the function may need to sleep, a Set Status call should be used rather than a Get Status call, to overcome this problem.

As with all I/O system calls, the Get Status (**I\$GetStt**) and Set Status (**I\$SetStt**) system calls go first to the kernel. The specific function required is indicated by a function code parameter to the system call (in the **d1.w** register). The kernel checks this code to see if it is known to the kernel. If so, the kernel executes the desired function. In either case, the kernel then calls the Get Status (or Set Status) routine of the file manager, passing it the same function code. If the file manager returns an "unknown request" error (**E\$UnkSvc**) for a function code that the kernel recognized, the kernel returns no error to the calling program. Otherwise the kernel returns the error returned by the file manager. Of course, if the kernel recognizes the function code, but experiences an error in executing the appropriate function, it does not call the file manager, but returns the error to the calling program.

Typically the file manager will behave like the kernel – that is, it checks the code, executes the appropriate function if it recognized the code, and then calls the Get Status (or Set Status) routine of the device driver. If the driver returns an "unknown request" error for a function code that the file manager recognized, the file manager returns no error to the kernel.

This mechanism whereby the call is passed from the kernel to the file manager, and from the file manager to the device driver, allows each of the three modules to implement any number of functions that may be unknown to the other two. And because even recognized calls are still passed down the tree, a call that requires action by two modules can be implemented. For example, the **SS_Opt** Set Status function to alter the options section of the path descriptor is acted on by the file manager, but because it is also afterwards passed to the device driver, the driver can use the new parameters in the options section to reconfigure the interface chip.

The kernel recognizes no Set Status function codes, and only two Get Status functions: **SS_Opt** (return a copy of the path descriptor options section, 128 bytes), and **SS_DevNm** (return a copy of the device descriptor module

name). These requests are made by the C library functions `_gs_opt()` and `_gs_devn()` respectively.

In addition to calls from a program, the file manager may generate Get Status or Set Status calls to the device driver. This is an alternative to defining additional routines in the driver, and has the advantage that the file manager writer can add more such calls in a later release of the file manager without the need to change the device driver – the driver will automatically return an "unknown request" error to the new calls. The kernel does not generate such calls, although it may do in future releases.

As with other system calls, the parameters are passed from the calling program in processor registers. However, because any function can be defined by the kernel, the file manager, or the device driver, the kernel or file manager cannot simplify the environment of the device driver by passing the parameters in process registers to the device driver. Instead, the driver must read the calling program's register stack frame (built by the kernel when the system call is made), which is pointed to by the `PD_RGS` field of the path descriptor. Similarly, to return values to the calling program the driver must write to the stack frame. For example, to read the calling program's `d2` register (in this case, into the device driver's `d0` register):

```
movea.l PD_RGS(a1),a5    get stack frame pointer
move.l  R$d2(a5),d0      get caller's d2 register
```

In C an equivalent code fragment would be:

```
x=pathdesc->pd_rgs->d[2];    /* get caller's d2 register */
```

The file `'DEFS/process.a'` defines the symbolic definitions – such as `R$d2` – for the structure of the stack frame in assembly language, while `'DEFS/MACHINE/reg.h'` declares the same for C. If the file manager itself generates a call that requires parameters, it must save the current parameter register values from the stack frame, put in the parameters it wishes to pass, call the device driver routine, and then restore the saved values. This complication is not needed if the call is one that will not be made from a program – it is only internally generated by the file manager. In this case the file manager could pass the parameters in processor registers, as is done for the read and write routines.

In general, a file manager will recognize at least the `SS_Opt` function of the Set Status call. This requests the file manager to update the options section of the path descriptor. The file manager implements this call, rather than the kernel, because normally the file manager will only permit the calling program to alter certain fields of the options section. For example, SCF will allow the program to modify any of the fields of the options section proper,

while RBF will only allow modification of fields up to and including the **PD_SAS** field.

SCF and RBF generate a Set Status call **SS_Open** to the device driver when a path is opened or created, so a new path descriptor has been created, and a Set Status call **SS_Close** when the last image of a path is closed (the path descriptor is about to be de-allocated). SCF also generates a Set Status call **SS_Relea** when process closes a path and the process does not have any remaining duplications of the path. This is done in case a process requested the sending of a signal when data was received (**SS_SSig** Set Status call to the driver), and then died without being sent the signal, and without having cancelled the request.

If the cancellation was not forced by SCF, when new data arrived the driver might send a signal to a new (and unsuspecting) process that was created with the ID released by the dead process. Note that the device driver writer must bear this kind of complication in mind when adding Set Status or Get Status functions to a device driver. A process that has installed a request for action at some future time may die unexpectedly in the intervening period, and this should not be catastrophic to the system. An SCF driver can use the **SS_Relea** and **SS_Close** calls from the file manager to ensure that all such pending requests for a process are cancelled. This requires that the driver save the process ID and system path number when the request is first made, so that it can match the pending request with the call from the file manager.

While device drivers will vary in the Get Status and Set Status functions that they support, a device driver for general use should support at least the functions that are supported by the standard device drivers supplied by Microware as example source code, and in the OS-9 implementations that Microware has carried out. This ensures a common base level environment that all programs can expect. The list below shows the standard functions for an SCF and an RBF device driver. The assembly language symbolic name for the function code is given, together with the C library function provided to make the call from a C program.

□ Get Status calls for an SCF Driver

<u>Name</u>	<u>C function</u>	<u>Description</u>
SS_Ready	_gs_rdy	Returns the number of characters available in the input buffer to the caller's d1 register. If there are no characters in the input buffer, returns "not ready" error - E\$NotRdy .

DEVICE DRIVERS

<u>Name</u>	<u>C function</u>	<u>Description</u>
SS_EOF	<code>_gs_eof</code>	Returns "end of file" error if at end of file. As SCF does not support a filing structure, there is no static end of file condition, so this function never returns an error in an SCF device driver.

The assembly language code below is typical of the Get Status routine of an SCF driver, and illustrates the use of the calling program's stack frame. Note that although the calling program passes the function code in the **d1.w** register, the file manager moves it to the **d0.w** register before calling the device driver. Of course, the function code could also be obtained from the **d1.w** register of the calling program's stack frame.

```

* GetStat
* SCF device driver Get Status "wild card" routine
* Passed:  d0.w = function code
*          (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor
*          (a6) = System Globals
* Returns: depends on function
*
GetStat:   movea.l PD_RGS(a1),a5    get caller's stack frame pointer
           movea.l V_PORT(a2),a3    get port (interface chip) address
           cmpi.w  #SS_Ready,d0     check data ready?
           bne.s   GetStat10        ..no
           move.l  InBufCnt(a2),d0   get number of characters in input
*                                     buffer
           beq.s   NotRdyErr         ..none; return "not ready" error
           move.l  d0,R$d1(a5)       return number in caller's d1
           bra.s   GetStatEx         ..exit; carry is clear

GetStat10  cmpi.w  #SS_EOF,d0        check for end of file?
           bne.s   UnkSvcErr         ..no; unknown request
* The carry flag is now clear. Fall through to exit - there is never an
* end of file condition on a terminal or printer:

GetStatEx  rts

* Return "not ready" error:
NotRdyErr  move.w  #E$NotRdy,d1      set error code in d1.w register
           ori     #Carry,ccr        set carry flag to show error
           rts

* Return "unknown request" error:
UnkSvcErr  move.w  #E$UnkSvc,d1      set error code in d1.w register
           ori     #Carry,ccr        set carry flag to show error
           rts

```

□ Get Status calls for an RBF Driver

Prior to OS-9 version 2.4 there were no standard Get Status calls for an RBF device driver. The following calls were added to RBF drivers in OS-9 version 2.4. As described with each call, drivers written prior to OS-9 version 2.4 – and so returning an "unknown request" error to these calls – will still work with OS-9 version 2.4, but without the benefit of some of the added features of OS-9 version 2.4.

<u>Name</u>	<u>C function</u>	<u>Description</u>
SS_VarSect	(none)	RBF makes this call when opening a path, to determine whether the driver and device can support sector sizes other than 256 bytes. If so, the driver should check the value in PD_SSize . If it is not zero, the driver should verify that it is a valid sector size, supported by the device and the driver. If so, the driver returns no error, otherwise the driver returns an error – typically "parameter error" – E\$Param – or "hardware error" – E\$Hardware . If PD_SSize is zero, the driver should put the current device sector size in PD_SSize , and return no error. If the driver does not support sector sizes other than 256 bytes (for example, most drivers prior to OS-9 version 2.4), the driver returns an "unknown request" error, in which case RBF assumes a sector size of 256 bytes, and ignores any value in PD_SSize . If the driver returns an error other than "unknown request" RBF aborts the opening of the path, otherwise RBF puts the sector size in PD_SctSiz . Note that RBF makes this request before allocating its sector buffers, so the driver cannot use the memory pointed to by PD_BUF .
SS_DSize	(none)	Request the disk data capacity in sectors. The format utility makes this call, to avoid the need to use the path descriptor options section parameters to calculate the disk data capacity if the controller can determine the capacity, thus allowing one form of device descriptor to be used for a range of disk drives. If the disk controller can determine the disk data capacity (that is, the disk space usable by the filing system, excluding sectors reserved by the controller or disk drive), the driver should issue a command to the controller to determine the disk capacity, and return it in the calling program's d2.l register (in the register stack frame). Otherwise the driver should return an "unknown request" error, in which case the format utility calculates the disk capacity from the number of cylinders, tracks per cylinder, and sectors per track specified in the path descriptor options section.

□ Set Status calls for an SCF Driver

<u>Name</u>	<u>C function</u>	<u>Description</u>
SS_Opt	_ss_opt	Modify the path descriptor options section. The device driver must check whether the parameters it uses to configure the device (such as PD_PAR and PD_BAU) have changed. If so, the driver must reconfigure the device. If there is no change, the driver should not reconfigure the device, as altering the configuration registers of an interface chip may corrupt characters currently being transmitted or received.
SS_SSig	_ss_ssig	Request that the driver send a signal to the process when data becomes available. The calling program passes the desired signal code in the d2.w register. The device driver must save not only the signal code, but also the caller's process ID, and the system path number (from the path descriptor field PD_PD), in order to know which process to send the signal to, and to provide a check for a subsequent SS_Relea call. Once the signal has been sent (usually by the interrupt service routine, when a character is received), the driver "forgets" the call. A new SS_SSig call must be made by the program if it wishes to receive another signal. If data is already available in the input buffer the driver must send the signal immediately. The driver must mask interrupts up to the interrupt level of the device while checking for input data (and perhaps sending a signal), to avoid a race condition with an incoming character invoking the interrupt service routine. Normally the device driver will permit only one such request to be pending at any one time. That is, if a process has made this request but has not yet be sent a signal (and has not cancelled the request with SS_Relea), then a "not ready" error - E\$NotRdy - is returned to any other SS_SSig request. Also, typically the driver will return a "not ready" error to any call to the read routine while an SS_SSig request is pending.
SS_DCOn	_ss_dcon	Request that the driver send a signal when the DCD (Data Carrier Detect) handshake input becomes asserted. This request (and the SS_DCOff , SS_EnRTS , and SS_DsRTS requests) is only appropriate for a serial port device with modem handshake lines (typically available on all asynchronous serial ports, such as RS232C). This request is similar to the SS_SSig request, except that the read routine does not return a "not ready" error if an SS_DCOn request is pending. This function can only be supported if the device can generate an interrupt when DCD becomes asserted (otherwise the driver must return an "unknown request" error - E\$UnkSvc).
SS_DCOff	_ss_dcoff	Request that the driver send a signal when the DCD (Data

<u>Name</u>	<u>C function</u>	<u>Description</u>
		Carrier Detect) handshake input becomes negated. This is similar to the SS_DCOff request. This function can only be supported if the device can generate an interrupt when DCD becomes negated (otherwise the driver must return an "unknown request" error - ES\$UnkSvc).
SS_Relea	<code>_ss_rel</code>	Cancels any outstanding SS_SSig , SS_DCOff , and SS_DCOff requests for this process on this path.
SS_EnRTS	<code>_ss_enrts</code>	Requests that the driver assert the RTS handshake output of the device. The circuit configuration or the interface chip behaviour may make it more appropriate to assert the DTR handshake output signal instead - the device driver documentation should make it clear which signal is manipulated by this function. If no output signal can be manipulated manually by the device driver it should return an "unknown request" error.
SS_DsRTS	<code>_ss_dsrts</code>	This is the complement of the SS_EnRTS request. It requests that the device driver negate the RTS (or DTR) handshake output of the device.

☐ Set Status calls for an RBF Driver

<u>Name</u>	<u>C function</u>	<u>Description</u>
SS_Reset	<code>_ss_rest</code>	Restore the drive head to cylinder zero. Floppy disk drives usually have a "head at cylinder zero" sensor. The only way in which the controller can know which cylinder the drive head is on is by knowing how many steps the head has taken from cylinder zero. After many steps backwards and forwards a slight positional error may accrue, due to the mechanical characteristics of the drive. The driver will therefore usually automatically use a special controller command to restore the head to cylinder zero when reading or writing a sector on cylinder zero, or after a read or write operation gives a "seek error" (in order to reconfirm that the head is on the correct cylinder). During a format operation, however, the controller cannot give a seek error, as it is not reading sector address information from the unformatted disk. Therefore the format utility makes this request after formatting a number of cylinders, to ensure the head alignment is correct (the driver will move the drive head to the correct cylinder on the next "format a track" request).
SS_WTrk	<code>_ss_wtrk</code>	Request to format a track of a disk. The calling program (usually the format utility) specifies a surface of a cylinder to format (see below).

DEVICE DRIVERS

The **SS_WTrk** request is used by the **format** utility, which issues a request to format each surface of each cylinder. The cylinder number (base zero) is specified in the caller's **d2.w** register. The surface number (base zero) is specified in bits 8:15 of the caller's **d3.w** register. Many controllers (such as SCSI controllers) have a command to format the whole disk. A driver for such a controller only reacts to a **SS_WTrk** request for track zero (both cylinder and surface are zero), in response to which the driver issues the controller command to format all of the disk. Such a driver takes no action and returns no error for requests on other cylinders or surfaces.

Many floppy disk control circuits use Western Digital controller chips, such as the 177x series, and the 279x series. These controllers require the computer to provide a complete byte stream to format the track. Because these controllers are in common use, and generating such a "track buffer" requires a great deal of detailed knowledge and programming effort, Microware has included the generation of a Western Digital track buffer in the **format** utility. The buffer is pointed to by the caller's **a0** register, and can be transferred directly to the controller chip in the same way as read or write data.

A driver for a different type of controller may need to build a track buffer or sector address list. This requires the use of the "physical sector interleave factor" to determine the order in which the sectors are to be numbered on the track. (This allows the optimization of reading or writing logically consecutive sectors). Because this is also a common requirement, the **format** utility builds a sector number list, otherwise known as an interleave table. The list is pointed to by the caller's **a1** register, and is the same for every track. It consists of an array of bytes, one for each sector number, in the order in which the sector numbers are to be used on the track. The sector numbers are base zero. Therefore the driver should add the offset in **PD_SOffs** to each sector number when building the track buffer or sector address list.

The third group of controllers comprises those (such as SCSI controllers) that require only a high level command to format a track or the whole disk. Such a controller may allow the driver to specify the physical sector interleave factor. The interleave factor is given in the **PD_ILV** field of the path descriptor options section. However, this field cannot be modified by the **SS_Opt** Set Status call. Therefore, in order to allow the user to specify the interleave factor using the '-i' option, the **format** utility passes the interleave factor in the **d4.b** register (using the value in **PD_ILV** if the '-i' option is not used). Therefore the driver should use the value in the caller's **d4.b** register, rather than the value in **PD_ILV**.

In order that the user can format a single track density disk in a double track density drive, the **format** utility also passes a field of format flags in the **d3.b** register. This field has the same structure as the **DD_FMT** field of LSN zero and the drive table, but applies separately for each track. For example, bit 0 is zero when formatting surface (side) zero, and one when formatting surface one (or other surfaces), rather than indicating whether the disk is single or double sided. To support disks with more than two surfaces, bits 8:15 of the **d3.w** register specify the surface number (base zero).

Some drivers (particularly those for "intelligent" controllers, such as SCSI controllers) also implement the **SS_DCmd** Set Status request, allowing the calling program to pass any command directly to the controller. There is no set parameter format for this request, and the calling program must know both the request parameter format and the controller command and response structure.

12.7 INTERRUPTS

This section discusses the purpose of interrupts, and how they are used under the OS-9 operating system. Although OS-9 makes no requirement that a device driver must use interrupts, they are essential to the proper operation of any multi-tasking or real time operating system. Interrupts are used for two distinct purposes:

- a) To signal the occurrence of hardware events for which there may be no process waiting. Examples are clock ticks, and serial port type-ahead and write-behind. I have named this an **unsolicited interrupt**, because the interrupt occurs without being specifically requested.
- b) To wake up a sleeping process that is waiting for the completion of a hardware operation, so allowing the processor to execute other processes. Examples are disk and tape operations. I have named this a **solicited interrupt**, because the interrupt cannot occur unless a process has requested the interrupt and is waiting for it.

Of course, unsolicited interrupts can only occur if the device driver (or other software) has enabled interrupt generation in the interface chip. Nonetheless, the distinction between solicited and unsolicited interrupts is an important one, with significant implications for the device driver writer.

DEVICE DRIVERS

Interrupts are a function of external hardware, and are therefore *totally asynchronous* to the normal program flow of control. It is most likely that the process waiting for the interrupt (if there is one!) is not executing at the time of the interrupt. It will be asleep, and another process will be executing. It is this asynchronicity that gives rise to all of the conceptual and programming problems of interrupts. Once this concept has been mastered the programming precautions necessary to use interrupts are obvious and simple.

An interrupt is handled by an interrupt service routine. Interrupt service routines are normally in device drivers, because it is the device drivers that handle the hardware that causes the interrupts. However, this is not a fixed requirement of OS-9. Any software operating in system state can install an interrupt service routine (using the **F\$IRQ** system call) – for example, a kernel customization module or a system state trap handler. The interrupt service routine itself can be located anywhere in memory, although it is normally located within the module that installed it. Otherwise there may be a risk that the module containing the interrupt service routine is unlinked before the corresponding interrupts are disabled.

The 68000 family of microprocessors supports 199 separate interrupt vectors – 7 autovectors (25 to 31) and 192 normal vectors (64 to 255). An OS-9 interrupt service routine services one such vector (although multiple interrupt service routines can be installed on the same vector). Because OS-9 is completely customizable, interrupt service routines can be dynamically installed and removed, using the **F\$IRQ** system call. At coldstart the kernel sets all of the exception jump table entries for interrupts to point to the kernel's interrupt handler function, and builds a table in the System Globals of 199 pointers, all initially null.

When the **F\$IRQ** system call is made to install an interrupt service routine, the kernel finds a free entry in the array of structures known as the interrupt polling table. It adds the entry to the linked list pointed to by the root pointer for the vector on which the interrupt service routine is being installed. Thus the interrupt polling table contains up to 199 separate linked lists, intertwined together. Within each linked list the order of the entries is determined by the "polling priority" value passed to the **F\$IRQ** call – a low priority value puts the entry nearer the root of the list. If two entries have the same priority value the chronologically later entry is placed after the earlier entry. A priority value of zero is a special case – the kernel ensures that this is the only entry on the specified vector. If the linked list ("queue") for this vector is not empty when the request with priority zero is made, the caller is returned a "vector busy" error (**E\$VctBsy**). The same error is

returned if an **F\$IRQ** call is made for a vector on which there is already an entry installed with a priority of zero.

The interrupt polling table entry contains the address of the interrupt service routine, the static storage pointer passed to the **F\$IRQ** call (usually the address of the device static storage), and the "port address" passed to the **F\$IRQ** call. When the processor initiates interrupt exception processing it jumps to the exception jump table entry for the interrupting vector, which jumps to the kernel's interrupt handler. The kernel uses the vector number to select the appropriate root pointer. It then calls the interrupt service routine from the first entry in the queue (lowest priority value). If that routine returns the processor carry flag set, the kernel calls the routine from the next entry, and so on until a routine returns the carry flag clear (or the queue is exhausted – see below).

This technique allows all 199 vectors to be supported without wasting memory. Multiple devices can use the same vector. However, this is normally only necessary for autovectored devices (the vector is determined by the interrupt level), as there are only 7 autovectors. The **F\$IRQ** system call is also used to remove an entry from the interrupt polling table, permitting complete termination of the resources of a device.

The installed interrupt service routine is called by the kernel with the **a2** register containing the same static storage pointer (and the **a3** register containing the same "port address") as was passed to the **F\$IRQ** system call. The static storage pointer will normally be the address of the device static storage for the interrupting device, allowing the device driver and interrupt service routine to have common access to shared variables. As the interrupt service routine is normally part of the device driver module, the interrupt service routine will use the same symbolic names for the variables as the main body of the device driver. The "port address" in the **a3** register is not used at all by the kernel – it is passed merely as a convenience to the interrupt service routine (which could otherwise have read it from the device static storage).

The interrupt service routine is only permitted to destroy the **d0**, **d1**, **a0**, **a2**, **a3**, and **a6** registers (unless bit 0 of the first compatibility byte in the **init** module is set). An interrupt service routine will not normally modify the processor interrupt mask in the status register, except perhaps to temporarily set the mask to level 7 to mask interrupts from other devices when executing code fragments that interact with other interrupting devices.

DEVICE DRIVERS

The interrupt mask should never be lowered below the interrupt level of the interrupting device, as this could lead to nested interrupts, eventually crashing the system. If the interrupt service routine cannot handle one interrupt from the device before it generates another interrupt, it will not help to expose the system to the second interrupt before the first has been handled! When the interrupt service routine finishes, it returns to the kernel with the **rts** instruction just like any other subroutine, not the **rte** instruction. In summary, the calling convention for an interrupt service routine is:

Passed: (a2) = static storage (usually device static storage)
 a3.1 = port address
 (a6) = System Globals
Returns: carry set if not this driver's interrupt
May destroy: d0-d1/a0/a2-a3/a6

While the kernel's interrupt handler does not make any use of the static storage pointed to by the interrupt polling table entry, the static storage pointer value is used to identify an entry in the linked list for a vector when the **F\$IRQ** call is used to remove an entry from the polling table. The kernel determines that the call is being used to remove an entry because the interrupt service routine address (in the **a0** register) is zero. It then scans the linked list for the given vector, looking for a match for the given static storage pointer (in the **a2** register). This implies that two entries on the same vector must not be installed with the same static storage pointer. This is not a problem – different device drivers (even different incarnations of the same device driver module) will have different device static storage addresses, and a driver will not need to install two interrupt service routines on the same vector.

An interrupt service routine cannot make use of most of the system calls. This is because the interrupt may occur while a process is making the same system call (or a related one), and a "nested" call might damage operating system memory structures. The following system calls are available for use by interrupt service routines (the kernel masks interrupts during critical code fragments in these system calls):

F\$Event	All event functions except Ev\$Creat , Ev\$Delet , Ev\$Link , and Ev\$Info .
F\$Send	Send a signal.
F\$AProc	Put a process into the active queue.
F\$NProc	Make the next process in the active queue the

	current process.
F\$Move	Copy a block of memory.
F\$CCtl	Flush, enable, or disable the processor caches.
F\$Time	Get the current date and time.
F\$Julian	Convert Gregorian date and time to Julian.
F\$Gregor	Convert Julian date and time to Gregorian.

Because of the asynchronous nature of OS-9 signals – able to cause the asynchronous execution of a signal intercept handler function – there is often conceptual confusion between interrupts and signals. The confusion is sometimes increased because most interrupt service routines send signals. Interrupts are a function of external hardware and the interrupt circuitry of the processor. Interrupts are masked using the interrupt mask field of the processor's status register. By contrast, signals are a software function only, and are masked by the **F\$SigMask** system call. If an interrupt service routine sends a signal, the receiving process's signal intercept handler is not called until the process next runs in user state, which cannot occur at least until the interrupt service routine has completed. The signal intercept handler is *not* called during the execution of the interrupt service routine, and interrupts are *not* masked when a signal intercept handler is called.

While the job to be done by an interrupt service routine varies widely, some basic principles apply. The interrupt service routine must first ascertain that its device caused the interrupt, usually by reading a status register from the interface chip. If not, it simply returns to the kernel with the processor's carry flag set. If the interrupt service routine was installed in the polling table with a priority of zero then it does not need to check that its device caused the interrupt, as it is the only device using this vector number. This is an essential mechanism for some interfaces that have no status flag showing that they have an interrupt pending.

Once the interrupt service routine has verified that its device generated the interrupt, it must:

- a) Clear the interrupt to the processor. Many normal vectoring devices clear the interrupt automatically once they have sent the interrupt vector to the processor (interrupt acknowledge cycle).
- b) Carry out any immediately required operations. These must be kept to a minimum – processes cannot run while an

interrupt is being serviced, and other interrupts on the same and lower interrupt levels cannot be serviced. In general, if at all possible operations should be left to be carried out by the device driver main body once it has been woken – which may incur a delay of tens of milliseconds.

- c) Wake up any waiting process. This refers to the main body of the device driver having executed a "sleep" request (**F\$Sleep**) on behalf of the calling process, waiting for the interrupt to occur. For a solicited interrupt there will always be a waiting process. For an unsolicited interrupt there may be a waiting process, but not always.

The handling of interrupts, as with most of the code in device drivers, is very much to do with understanding and managing the hardware. However, a discussion of the control of hardware interface devices is outside the scope of this book. From an operating system point of view the important element is the interaction with any waiting process. It is with this aspect that the following discussion is concerned.

12.7.1 Solicited Interrupts

A solicited interrupt should be used wherever the device driver estimates that a hardware operation will take longer than the time that would be required for the driver to go to sleep and be woken by an interrupt service routine. That is, more processor time will be used by polling the interface status register until the operation is complete than by waiting for an interrupt.

Solicited interrupts are relatively easy to handle. The device driver decides that a hardware operation is going to take some time, and rather than wait by polling a status register it elects to give up its usage of processor time and wait for an interrupt. As the interrupt cannot occur until the driver has performed the device function that initiates the interrupt mechanism, control is straightforward:

- 1) Set a flag in the static storage indicating to the interrupt service routine that a process needs waking, together with the ID of the process to wake (the current process). It is usually convenient to combine the two items, because no process has a process ID of zero. Therefore if the static storage field containing the ID of the process to wake is zero, no process is

waiting to be woken.

- 2) Initiate the device operation, with the interface chip set to generate an interrupt when the operation is complete.
- 3) Go to sleep. The **F\$Sleep** system call will return when the process is woken by a signal, or – for a timed sleep – when the sleep time expires. A timed sleep is only used if the driver wishes to implement a timeout on the hardware operation.

It is very important not to reverse operations 1 and 2. The interrupt may come in at any time *after* the device operation has been initiated, and the interrupt service routine must know that it has a process to wake. It does not matter if the interrupt occurs between stages 2 and 3 (that is, before the driver has executed the "sleep" request). The kernel leaves the signal sent by the interrupt service routine pending in the process descriptor. The **F\$Sleep** system call sees that a signal has been received and immediately returns to the driver without putting the process to sleep.

Once the driver has been woken it must verify that the interrupt service routine sent the signal – the signal may have come from another process communicating with the process that called the device driver. If the hardware operation is not complete the driver must go back to sleep (unless it decides that the received signal was "deadly"). Because the driver is executing in system state, all the signals sent to the process are queued in the process descriptor until the process returns to user state (at the end of the system call that called the driver). Therefore no signals are lost. The "wakeup" signal – **S\$Wake** – is an exception. It is not queued, and is therefore only suitable for use by an interrupt service routine waking up a device driver.

The driver is woken by each signal received. The kernel sets a flag in the process descriptor to show that the latest signal caused a wakeup, so that when the driver goes back to sleep (because the signal was not from the interrupt service routine), the **F\$Sleep** system call permits the sleep – it does not return immediately to the driver, even though a signal is pending for the process.

Because this mechanism is so commonly used, Microware have defined two fields in the kernel part of the device static storage to support it: **V_BUSY** and **V_WAKE**. These fields are not used at all by the kernel. The field **V_BUSY** contains the ID of the calling process, set by the file manager as part of its interlock on the device (see the section on Resource Control in the chapter on File Managers). The field **V_WAKE** is the flag field described

DEVICE DRIVERS

above. The driver copies the process ID to this field, setting it non-zero as an indication to the interrupt service routine that a process needs waking. The interrupt service routine clears the field (after taking the process ID) as a handshake to the main body of the driver, and to prevent further wakeups. For example:

	<code>move.w</code>	<code>V_BUSY(a2),V_WAKE(a2)</code>	set flag and process ID
	<code>bsr</code>	<code>IssCmd</code>	initiate device operation
Loop	<code>moveq</code>	<code>#0,d0</code>	indicate indefinite sleep
	<code>os9</code>	<code>F\$Sleep</code>	sleep until woken
	<code>tst.w</code>	<code>V_WAKE(a2)</code>	woken by interrupt?
	<code>bne.s</code>	<code>Loop</code>	..no; go back to sleep

In this example the driver does not consider any signal is "deadly" – that is, a signal important enough to abort the operation. Therefore if on wakeup it finds that it has not been woken by the interrupt service routine, it goes back to sleep without checking the signal that caused the wakeup.

Note the use of the `V_BUSY` field as the source of the process ID. Most file managers put the current process ID in this device static storage field. However, the kernel does not set this field, and so it is not valid during the initialization and termination routines. A driver that needs to use interrupts within the initialization or termination routines must take the process ID from the process descriptor:

	<code>move.w</code>	<code>P\$ID(a4),V_WAKE(a2)</code>	set flag and process ID
--	---------------------	-----------------------------------	-------------------------

If a common "sleeping" subroutine is used that assumes `V_BUSY` contains the process ID, then the initialization and termination routines could copy the process ID to the field. However, the initialization routine must be sure to clear this field before exiting, as the file manager will expect it to be clear in subsequent I/O calls (see the chapter on File Managers). Note that it is in any case inadvisable to sleep within the initialization routine (see the preceding section on the Initialize routine).

The corresponding interrupt service routine would be as shown below, assuming the routine has already determined that this is its interrupt, and taken any necessary action to clear it:

	<code>move.w</code>	<code>V_WAKE(a2),d0</code>	get ID of process to wake
	<code>beq.s</code>	<code>IRQExit</code>	..none; (should not happen)
	<code>clr.w</code>	<code>V_WAKE(a2)</code>	show valid interrupt wakeup
	<code>moveq</code>	<code>#S\$Wake,d1</code>	send special wakeup signal
	<code>os9</code>	<code>F\$Send</code>	send the signal
IRQExit	<code>moveq</code>	<code>#0,d1</code>	clear carry - interrupt serviced
	<code>rts</code>		return to kernel

Note the use of the signal code **S\$Wake**. As already described, the kernel assigns special properties to this signal code, so that its only function is to ensure that a process is in the active queue.

12.7.2 Unsolicited Interrupts

Unsolicited interrupts – such as from serial port received data – are slightly more complex to handle. The device may generate an interrupt at any time, so it is important to prevent timing race conditions between the interrupt service routine and the main body of the device driver. This is done by preventing the recognition of the interrupt by the processor during critical code fragments in the main body of the driver. To do this, interrupts are masked in the status register up to the interrupt level of the device.

The interrupt level of the device is specified in the **M\$IRQLvl** field of the device descriptor. The initialization routine of the driver can build a status register image with the interrupt mask set to that level, and save it in the device static storage for later use:

```

move.b    M$IRQLvl(a1),d0  get device interrupt level
lsl.w     #8,d0             shift to bits 8:10
bset      #SupvrBit+8,d0    set supervisor state bit
move.w    d0,IRQMask(a2)   save sr image

```

The following example is typical of a serial port device driver read routine. For simplicity this example ignores the need to send the XON flow control character if XOFF had been sent and the buffer is now at the low water mark:

```

Read      tst.w    SigPrc(a2)      SS_SSig request pending?
          bne      NotRdyErr       ..yes; read request not allowed
          move     sr,-(a7)         save current interrupt mask
          move     IRQMask(a2),sr   mask interrupts to device level
          bsr      InBufOut         get character from input buffer
          bcc.s    Read20          ..got one (in d0.b)

* The input buffer was empty. Sleep, waiting for data:
          move.w    V_BUSY(a2),V_WAKE(a2)  set flag and process ID
          move     (a7)+,sr         unmask interrupts
          bsr      Sleep            sleep
          bcs.s    ReadEx           ..fatal signal received; abort
          bra.s    Read            ..else try again

Read20    move.b    V_ERR(a2),d1     get error flag
          clr.b     V_ERR(a2)       reset it
          move     (a7)+,sr         unmask interrupts
          tst.b     d1              any errors?
          beq.s     ReadEx          ..no; carry is clear
          move.w    #E$Read,d1      return read error
          ori      #Carry,ccr       set carry to show error

```

DEVICE DRIVERS

```
ReadEx      rts
```

```
* Read request made while SS_SSig request is pending:
NotRdyErr   move.w  #E$NotRdy,d1    return "not ready" error
            ori     #Carry,ccr
            rts
```

The **InBufOut** subroutine gets a character from the input circular buffer, returning it in the **d0.b** register. If the input buffer is empty, the subroutine returns the processor carry flag set.

The **Sleep** subroutine sleeps indefinitely until woken by a signal, and then checks whether a deadly signal has been received by the process, or the process has been condemned (sent a "kill" signal). If so, the driver exits immediately with the signal code as the error code (or 1 if the process is condemned), not waiting to complete the I/O operation. (This would not be suitable in an RBF driver, where the operation must be completed or the disk filing system may be corrupted.)

Note that prior to OS-9 version 2.4 the **P\$Signal** field of the process descriptor contained the *oldest* pending signal (the next signal to be processed). Therefore if a non-deadly signal was received followed by a deadly signal the check in the driver would only see the non-deadly signal, and not abort. From OS-9 version 2.4 onwards the **P\$Signal** field contains the most recently received signal (not yet processed by the user program), so by checking this field the driver will see each signal in turn. Also, prior to OS-9 version 2.4 only the abort (quit) and interrupt signals (2 and 3) were considered deadly. From OS-9 version 2.4 onwards all signal codes below 32 are considered deadly.

```
Sleep        moveq   #0,d0          sleep without timeout
              os9     F$Sleep
              move.w  P$Signal(a4),d1 get most recent signal in d1.w
              beq.s   Sleep10        ..none
              cmpi.w  #S$Deadly,d1   deadly signal?
              bcs.s   SleepEr        ..yes; error
Sleep10       moveq   #0,d0          ensure carry is clear
              btst    #Condemn,P$State(a4) is process dead?
              beq.s   SleepEx        ..no; exit with carry clear
              moveq   #1,d1          "unconditional abort" error
SleepEr       ori     #Carry,ccr     set carry to show error
SleepEx      rts
```

The corresponding code fragment in the interrupt service routine to wake up the waiting driver is the same as for a solicited interrupt. However, while for a solicited interrupt device driver it would be an error for an interrupt to occur without there being a process to wake up, in the case of unsolicited interrupts this is a common occurrence.

The important point to note in the above read routine example is that the processor interrupt mask was set to the interrupt level of the device while the check on the input buffer was made, and interrupts were not unmasked until a character had been taken from the buffer or the wakeup handshake flag **V_WAKE** had been set. Remember also that the processor automatically sets its interrupt mask to the interrupt level of the device during an interrupt service routine. The result is that these two code fragments are mutually exclusive – neither can asynchronously break into the other – permitting an "indivisible" set of operations. This does not preclude a higher level interrupt from another device being serviced while either routine is executing, but as that interrupt service routine is not communicating with this driver the possibility is not relevant. (If the device driver *does* also service a higher level interrupt, critical code fragments should mask interrupts to the higher level).

Note that once the **V_WAKE** flag is set the driver can unmask interrupts (indeed, it must unmask interrupts before making the **F\$Sleep** system call). If an interrupt comes in after **V_WAKE** is set but before the driver has gone to sleep, the interrupt service routine will still send the signal. Because the process is the current process (it is not yet in the sleeping queue), the kernel will set the **B_WAKEUP** bit of the **P\$SigFlg** field of the process descriptor, and the subsequent **F\$Sleep** system call will return to the driver without sleeping.

Similarly, because the interrupt service routine clears the **V_WAKE** field when sending the signal, on wakeup the device driver will find this field clear if it has been sent the signal (although in this example the driver does not use this flag, but instead checks the input buffer again). Provided the device driver writer takes care to provide such an indivisible handshake between the main body of the driver and the interrupt service routine, there is no possibility of a timing race condition, and no interrupts will be missed.

The write routine for a serial port device driver is almost identical to the read routine, except that the write routine must sleep if the output buffer is full when it tries to put a character into the buffer. Also, the write routine has the responsibility for starting the "transmit stream" if transmitter interrupts had been disabled because the output buffer was empty.

When the interface chip generates a "transmitter ready" interrupt, the interrupt service routine checks the output buffer. If the buffer is not empty the interrupt service routine takes the next character from the buffer and writes it to the transmit register of the chip. The chip will generate another interrupt when its transmit register becomes empty again. Thus a continuous

stream of interrupts and character transmissions is maintained so long as the output buffer is not empty, which will be the case so long as the program (and SCF and the driver) provides data faster than the data transmission rate of the interface. If the buffer is empty, the interrupt service routine must command the chip to disable further "transmitter ready" interrupts, and set a "transmitter interrupts disabled" flag in the device static storage. Note that at this time the chip has room for at least one character in its transmitter register.

Before attempting to put the character in the output buffer (but after masking interrupts), the write routine checks whether the buffer is empty and transmitter interrupts are disabled (transmitter interrupts could be disabled because the "data received" interrupt service routine received the XOFF flow control character). If so, it knows the transmit stream has been broken, and must be restarted. It does this by writing the character directly to the transmit register of the chip (rather than to the output buffer in the device static storage), and enabling transmitter interrupts from the chip. It then clears the "transmitter interrupts disabled" flag.

Whether the write routine writes the character to the transmit register and then enables transmitter interrupts in the chip, or vice versa, depends on the behaviour of the transmitter interrupts of the chip. It is more widely applicable to enable the interrupts first, and then write the character. This will work if the chip generates an interrupt so long as the transmit register is empty (the interrupt will be generated, but then cleared when the character is written – meanwhile, the write routine has interrupts masked in the processor). It will also work if the chip generates an interrupt when the transmit *becomes* empty, provided the transmitter interrupts are enabled at that time. Enabling the interrupts before writing the character avoids a potential race condition.

12.7.3 Choosing Interrupt Levels

There has frequently been a much confusion over the philosophy which should be used to decide what interrupt level to assign to each device. However, a little thought will show that the decision can be made very easily. The only benefit of assigning a higher level of interrupt to one device than to another is that interrupts from the first device will pre-empt the service of interrupts for the second, and be accepted by the processor when the device driver for the second device has interrupts masked to the level of its device.

As all interrupts must eventually be handled by the processor, the important concern is that the interrupt from a device must be handled before the device

wishes to generate another interrupt of the same type. For example, if a serial port chip generates a "data received" interrupt, it must be serviced – and the character read from the chip – before the chip receives another character, assuming the chip has only a single level of buffer for received characters in addition to its receive shift register. However, if the chip has an 8 byte FIFO for received characters, it does not matter if the interrupt is not serviced before another character is received, provided it is serviced before 8 characters are received.

One important point is apparent here – solicited interrupts almost never need to be on a high level interrupt. Such interrupts are only generated in response to a command from the driver to the chip. If the driver takes a long time responding to the interrupt, no problem is caused, because the chip cannot need to generate another interrupt until the driver issues another command. Therefore chips that only generate solicited interrupts can be on a low interrupt level – 1 or 2, for example. A high interrupt level is only needed if a remote device needs a rapid response. For example, a communications protocol may specify a maximum response time.

This only leaves the question of how to select the interrupt levels for devices that generate unsolicited interrupts, such as communications ports, the clock tick hardware, and some network interfaces. Again, the answer is simple. The highest level of interrupt should be assigned to the device that can produce the shortest interval from one interrupt to the next. For example, a serial port operating at 19200 baud will generate interrupts roughly every 500 μ s, whereas a typical clock tick is 10ms. It follows that unless the serial port has a 20 character FIFO (unlikely!), it should have a higher level interrupt than the tick hardware. That is to say, it is less important that the response to a tick interrupt be delayed by a few microseconds, than that the serial port interrupt response be delayed by a similar time.

The only modifying consideration is the seriousness of the loss of an interrupt from a particular device. For example, a lost serial port interrupt will cause a communications error – hopefully recoverable – while a lost tick interrupt will cause an unrecoverable date and time error. However, a system that is so heavily loaded with interrupts is probably on the edge of failing in the application in any case.

12.8 A SKELETON DEVICE DRIVER

Often, part of the problem in writing a device driver for OS-9 is in knowing how to start. To help overcome this difficulty, this section shows the skeleton

DEVICE DRIVERS

of a device driver in assembly language. It provides the bones on which a device driver that actually controls a hardware interface can be built. Note the use of the file '/dd/DEFS/oskdefs.d'. This file contains definitions – such as module types – that cannot conveniently be taken from a library, due to limitations in the linker on the use of external symbols in arithmetic expressions.

```
* Skeleton device driver
Typ_Lang    set      (Drivr<<8)+Objct    module type and language
Att_Revs    set      ((ReEnt+SupStat)<<8)+0  module attributes and
*                                                  revision
Edition     set      1                    software edition number
            psect    skeldrv,Typ_Lang,Att_Revs,Edition,0,EntryTable
            use      /dd/DEFS/oskdefs.d

* Static storage definitions (to form the last part of the Device
* Static storage):
            vsect
IRQMask     ds.w      1                    sr image with interrupts masked
            ends      end of static storage definitions

* Routine offset table:
EntryTable   dc.w      Init                initialize
            dc.w      Read                 read
            dc.w      Write               write
            dc.w      GetStat             get status
            dc.w      SetStat             set status
            dc.w      Term                terminate
            dc.w      0                   (exception handler)

* Initialize
* Passed:    (a1) = Device Descriptor
*            (a2) = Device Static Storage
*            (a4) = Process Descriptor of current process
*            (a6) = System Globals
* Returns:   carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a6,ccr
Init         tst.w     d0                    clear carry - no error
            rts

* Terminate
* Passed:    (a1) = Device Descriptor
*            (a2) = Device Static Storage
*            (a4) = Process Descriptor of current process
*            (a6) = System Globals
* Returns:   carry set if error, with error code in d1.w
* (kernel ignores any returned error)
* May destroy: d0-d7/a0-a5,ccr (NOT a6)
Term         tst.w     d0                    clear carry - no error
            rts
```

```

* Read
* Passed:  (a1) = Path Descriptor - (NOT SBF)
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor of current process
*          (a6) = System Globals
*   RBF only:  d0.l = number of sectors to read
*   RBF only:  d2.l = LSN of first sector to read
*   SBF only:  d0.l = number of bytes to read
*   SBF only:  (a0) = buffer to read to
*   SBF only:  (a3) = drive table
* Returns:  carry set if error, with error code in d1.w
*   SCF only:  d0.b = character read
*   SBF only:  d1.l = number of bytes read
* May destroy: d0-d7/a0-a6,ccr
Read      tst.w  d0              clear carry - no error
          rts

* Write
* Passed:  (a1) = Path Descriptor - (NOT SBF)
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor of current process
*          (a6) = System Globals
*   RBF only:  d0.l = number of sectors to write
*   RBF only:  d2.l = LSN of first sector to write
*   SCF only:  d0.b = character to write
*   SBF only:  d0.l = number of bytes to write
*   SBF only:  (a0) = buffer to write from
*   SBF only:  (a3) = drive table
* Returns:  carry set if error, with error code in d1.w
*   SBF only:  d1.l = number of bytes written
* May destroy: d0-d7/a0-a6,ccr
Write     tst.w  d0              clear carry - no error
          rts

* Get status
* Passed:  (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor of current process
*          (a6) = System Globals
*          d0.w = function code
* Returns:  carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a6,ccr
GetStat   move.w  #E$UnkSvc,d1    unknown code
          ori     #Carry,ccr      return error
          rts

```

DEVICE DRIVERS

```
* Set status
* Passed:  (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor of current process
*          (a6) = System Globals
*          d0.w = function code
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a6,ccr
SetStat    move.w    #E$UnkSvc,d1    unknown code
           ori      #Carry,ccr      return error
           rts

           ends                    end of code
```

12.9 CLOCK DRIVERS

Each OS-9 system must have "clock" hardware and a clock driver, to support time-sliced multi-tasking, timed sleeps, alarms, and the maintenance of the date and time. As a minimum, the clock hardware must have a periodic timer, generating interrupts at regular intervals. This is the system "tick" interval, usually 10ms. The interval between ticks must be precisely constant, as the ticks are used to maintain the system date and time. This implies that the timer must be cyclic – there must be no need for software to retrigger the timer. Also, the tick period must be such that there are an integral number of ticks per second.

The clock hardware may also include a battery-backed "time of day" circuit. This is typically a separate chip, maintaining the date and time even when the computer is switched off.

It is the job of the clock driver to control this hardware, and to call the kernel's tick handler when a tick interrupt occurs. The clock driver is different from the OS-9 I/O device drivers. There is no associated device descriptor, path descriptor, device static storage, or file manager, and there can only be one clock driver in each system. The clock driver does not have a routine offset table such as device drivers have. Instead, the execution entry offset in the module header (the **M\$Exec** field) points directly to the initialization routine of the clock driver. The kernel takes the name of the clock driver module from the **init** configuration module. The clock driver name string is pointed to by the offset in the **M\$Clock** field of the **init** module.

The **F\$STime** system call is used to set the date and time. The kernel's handler for this function writes the new date and time to the System Globals (**D_Year**, **D_Month**, **D_Day**, **D_Second**, and **D_Julian** fields), and then

calls the initialization routine of the clock driver. The clock driver must ensure that the tick hardware is configured and running, and write the date and time to the battery-backed clock chip (if the driver supports one). Because the **F\$STime** system call may be made more than once, the driver should check whether it has already initialized the tick hardware. If so, it should not re-initialize it. The clock driver's initialization routine is called with the following parameters:

- (a4) = Process Descriptor of calling process
- (a5) = Caller's register stack frame
- (a6) = System Globals

The initialization routine may destroy any registers except **a4**, **a5**, and **a6**. If the routine encounters an error, it should return it in the normal way – the carry flag set, and an error code in the **d1.w** register. When initializing the tick hardware, the driver must perform three functions:

- a) Install its tick interrupt handler, using the **F\$IRQ** system call. As there is no clock device descriptor, the driver must use hard-coded values for the interrupt vector and software polling priority (and the interrupt level). Although when making the call the **a2** register must not match that for any other device installed on the same vector, this is not normally a problem for the clock driver, as the kernel passes **a2** pointing to the module directory entry for the clock driver. However, if the clock driver uses **a2** before making the **F\$IRQ** call, the driver writer must ensure it cannot be equal to the device static storage address of any present or subsequently installed device. Setting **a2** to zero for the **F\$IRQ** system call is therefore recommended by Microware as being safe and consistent.
- b) Initialize the **D_TckSec** (ticks per second) field of the System Globals. It is the responsibility of the clock driver to determine the number of ticks the tick hardware will generate each second. This keeps the kernel independent of the tick hardware. Any tick rate is permissible, provided that there is an integral number of ticks per second. 10ms is a typical period, giving 100 ticks per second. Too small a tick interval will cause tick interrupts and process scheduling to consume too large a fraction of the processor's time. Too large a tick interval may delay the real-time response of processes in a multi-tasking application, and may give too coarse a resolution for timed sleeps and alarms.

- c) Initialize the tick hardware, including enabling the tick interrupts, provided the hardware has not been initialized by a previous call to the clock driver's initialization routine.

Because the driver must set the **D_TickSec** field of the System Globals, and the kernel initializes this field to zero, the driver can use this field to check whether its initialization routine has been called before – **D_TickSec** will be zero if the initialization routine is being called for the first time.

Having initialized the tick hardware if necessary, the clock driver must write the new date and time to the battery-backed clock chip, if one is supported. The time and date are in the caller's register stack frame. **R\$d0(a5)** gives the time as 00HHMMSS, and **R\$d1(a5)** gives the date as YYYYMMDD (as required by the **F\$STime** system call).

However, if the month and day are zero, this is a request to read the date and time from the battery-backed clock chip, if one is supported. Instead of writing to the chip, the driver must read the current date and time from the chip, and set the **D_Year**, **D_Month**, **D_Day**, **D_Second**, and **D_Julian** fields of the System Globals. The **F\$Julian** and **F\$Gregor** system calls can be used to translate between Gregorian and Julian date and time formats. Note that the **D_Second** field is the number of seconds left until midnight, rather than seconds since midnight.

The kernel makes the **F\$STime** system call with a date of zero as part of its coldstart procedure (after the calls to open the default paths, change the directories to the default mass storage device, and install the kernel customization modules, if any), unless bit 5 is set in the first compatibility byte of the **init** module. In this way the kernel starts the clock, and reads the current date and time if a battery-backed clock chip is supported by the clock driver.

The interrupt service routine of the clock driver is usually straightforward. As with any interrupt service routine, it must determine that the interrupt was generated by the tick hardware, and return to the kernel with the carry flag set if not. Otherwise it must clear the tick interrupt in the tick hardware (if it is not automatically cleared by the interrupt acknowledge cycle), and call the kernel's tick handler. The address of the kernel's tick handler is in the **D_Clock** field of the System Globals:

```
movea.l D_Clock(a6),a0    get tick handler address
jmp      (a0)              ..go to it
```


Just like any other interrupt service routine, the clock driver interrupt service routine may only destroy the **d0**, **d1**, **a0**, **a2**, **a3**, and **a6** registers (unless bit 0 of the first compatibility byte in the **init** module is set).

In most systems the clock tick interrupts are not produced by the same chip that provides the battery-backed date and time facility, but by a separate timer chip. To simplify the job of the clock device driver writer, from OS-9 version 2.3 onwards Microware's example clock drivers are separated each into two source files. One file contains the routines for managing the clock tick chip, and the other contains the routines for managing the date and time chip. The files have a common interface, so the clock driver can be made for any combination of the two chips.

CHAPTER 13

FILE MANAGERS



To avoid the unnecessary duplication of many paragraphs, this chapter assumes that you have already read the chapter on Device Drivers.

File managers are essential components of the OS-9 I/O system. An understanding of existing file managers allows the programmer to make the most effective use of the I/O system within an application. However, some applications are best served by writing a new file manager. This chapter aims to describe the purpose of a file manager, and its interface to the kernel and device drivers, to give the system programmer the essential information needed to write a file manager. The Microware RBF and SCF file managers are described as examples, and to help the application programmer make best use of the I/O system.

The descriptions and code fragments in this chapter assume that the file manager is written in 68000 assembly language. However, file managers may equally well be written in C. The chapter on "Microware C and Assembly Language" describes how this is done.

13.1 THE FUNCTION OF A FILE MANAGER

Under OS-9, a file manager performs the filing structure maintenance and data processing for a class of like devices. That is, it performs all the logical operations on the devices. The file manager does not know how to control the hardware. Instead, it makes calls to the appropriate device driver to perform low-level hardware operations. This division of functionality allows one file manager to be used to manage a wide range of devices of a similar type, including devices not existent when the file manager was written. For

FILE MANAGERS

example, the Microware Random Block File manager (RBF) provides a hierarchical filing structure on almost any rewritable block structured device, while the Microware Sequential Character File manager (SCF) is suitable for almost all terminals and printers. Other file managers are available from Microware and third-party suppliers.

In addition, the file manager writer is able to concentrate on the problems of logical data manipulation, while the device driver writer handles the complexities of interrupts and VLSI interface chips.

Because one file manager is designed to work with many – as yet unwritten – device drivers, the file manager writer is also responsible for producing the specifications for:

- a) The functions and parameter conventions of the device driver routines (other than initialize and terminate).
- b) The options section of the path descriptor and device descriptor.
- c) The file manager storage in the path descriptor.
- d) The file manager storage in the device static storage, including the drive tables (if the file manager supports multiple channels on one device).

While the kernel prevents concurrent accesses on the same path, it is normally the file manager's responsibility to prevent concurrent accesses to the same device. The kernel will queue an I/O call on a path on which there is currently an I/O call being executed by another process. That is, two processes have the same path (the same system path number) open (because one inherited it from the other, or they both inherited it from the same parent or other ancestor). One process has made an I/O call on the path, and been put to sleep during the call by the file manager or device driver (usually only the device driver will sleep, waiting for a device operation to complete). The second process makes a call on the same path. The kernel will queue (put to sleep) the second process until the I/O call of the first process finishes.

The kernel uses the **F\$IOQu** system call, which puts the process to sleep, and links its process descriptor onto a linked list of process descriptors rooted in the process descriptor of the process using the path. This creates a "queue" of processes waiting for the process at the root of the queue to finish its I/O operation. In this way the kernel prevents the use of one path descriptor by two processes simultaneously. Note that no queuing is required if the first

I/O call does not sleep, as rescheduling is suspended while a system call is being executed, so the second process would not have the opportunity to make its I/O call until the first call finished.

However, the kernel does not check whether an I/O call on a *different* path is to the same device – perhaps even on the same file – as a currently executing call. This is because the kernel makes no assumption that the device cannot handle multiple requests simultaneously. While the handling of multiple hardware transactions concurrently is really a device driver function, most file managers assume that the device cannot handle more than one transaction at a time. Such file managers therefore prevent concurrent calls into a particular device driver incarnation, by queuing calls that they wish to make to the device driver until the device driver has finished executing any current request. A typical mechanism for doing this is described in the section on Resource Control.

13.2 FILE MANAGER ROUTINES

Each file manager module provides a set of routines to carry out I/O functions on a path. The file manager routines are only called by the kernel, in response to I/O system calls. Therefore the number, basic function, and parameter convention of these routines is fixed. However, because the kernel is not concerned with filing structures and data processing, the detail of the function of each routine may vary significantly between file managers.

The file manager routines correspond directly to the OS-9 I/O system calls:

<u>System call</u>	<u>Function</u>	<u>Description</u>
I\$Create	Create	Create a new file, and open a path to it.
I\$Open	Open	Open a path to an existing file or device.
I\$MakDir	Make directory	Create a new directory.
I\$ChgDir	Change directory	Change the current data and/or execution directory of the calling process.
I\$Delete	Delete	Delete a file.
I\$Seek	Seek	Set the file pointer of an open path (the position within the file for the start of the next read or write).
I\$Read	Read	Read data without editing.
I\$ReadLn	Read line	Read data, perhaps with line editing, ending on Carriage Return or other end-of-record character.
I\$Write	Write	Write data without editing.

FILE MANAGERS

<u>System call</u>	<u>Function</u>	<u>Description</u>
I\$WritLn	Write line	Write data, perhaps with line editing, ending on Carriage Return or other end-of-record character.
I\$GetStt	Get Status	"Wild card" call to get information about a device or path.
I\$SetStt	Set Status	"Wild card" call to send information to or request action on a device or path.
I\$Close	Close	Close a path.

A file manager can choose not to implement one or more functions, returning (with or without error) directly to the kernel, or perhaps passing the call without interpretation to the device driver. This is particularly the case for the file-related functions – Make Directory, Change Directory, Delete, and Seek – as these are not appropriate to certain types of device (those that cannot support a filing system), and for unrecognized sub-functions of the Get Status and Set Status calls.

13.3 KERNEL ACCESS TO THE FILE MANAGER

The kernel accesses the file manager routines by means of a table of offsets to the routines, similar to that for device drivers. The offset to the table from the start of the module header is in the "execution offset" entry (**M\$Exec**) of the module header. Unlike the device driver offset table, however, the entries are relative to the start of the table, not the start of the module. For example:

EntryTable	dc.w	Create-EntryTable	create a file
	dc.w	Open-EntryTable	open a file
	dc.w	MakDir-EntryTable	make a directory
	dc.w	ChgDir-EntryTable	change default directory
	dc.w	Delete-EntryTable	delete a file
	dc.w	Seek-EntryTable	set the file pointer
	dc.w	Read-EntryTable	read data
	dc.w	Write-EntryTable	write data
	dc.w	ReadLine-EntryTable	read data with line editing
	dc.w	Writeln-EntryTable	write data with line editing
*			
	dc.w	GetStat-EntryTable	get information
	dc.w	SetStat-EntryTable	send a command
	dc.w	Close-EntryTable	close a path

After each call to the file manager, if the I/O queue (**F\$IOQu**) rooted in the process descriptor of the calling process is not empty, the kernel marks the calling process as "timed out", forcing a reschedule when the system call finishes. (In OS-9 version 2.2 the calling process was marked as timed out

even if the I/O queue of the calling process was empty.) The kernel also performs an "I/O unqueue" operation. That is, if the I/O queue of the calling process is not empty, the calling process is detached from the queue, and the first process in the queue is woken up. This allows other processes to get a chance at the path or device, rather than letting the same process make another request and "hog" the path or device. It also improves the throughput of the I/O device – often the bottle-neck in a system – by quickly giving time to a process that wants to use the device, rather than allowing the previous process to finish its time slice.

13.4 PARAMETER CONVENTION

The kernel calls all the file manager routines with the same parameters:

- (a1) = Path Descriptor
- (a4) = Calling process's Process Descriptor
- (a5) = Caller's register stack frame
- (a6) = System Globals

The kernel sets up the following path descriptor locations before calling the file manager:

PD_CPR	Process ID of the calling (current) process
PD_LProc	Same as PD_CPR
PD_RGS	Address of caller's register stack frame (same as a5)

Note that the file manager is always supplied a properly initialized path descriptor. The kernel allocates and initializes a new path descriptor before calling the Create, Open, Make Directory, Change Directory, and Delete routines of the file manager. The initialization of the path descriptor includes attaching the device (**I\$Attach**), even if the pathlist does not start with a "/" (that is, the pathlist is relative to the data or execution directory). The kernel copies the device descriptor options section to the path descriptor options section.

Immediately after calling the Make Directory, Change Directory and Delete file manager routines, the kernel terminates the path by de-allocating the path descriptor and executing an **I\$Detach** system call, because these calls do not return an open path to the calling program. The kernel also terminates the path after calling the Close routine of the file manager if the use count of the path is zero.

The kernel expects the file manager to preserve the **a5** and **a6** registers, and the high byte of the status register. The file manager may destroy the other data and address registers.

13.5 PATHLISTS

A pathlist is the principal parameter to the Create, Open, Make Directory, Change Directory, and Delete routines. If the pathlist begins with the '/' character, the kernel only interprets the first name element, assuming it to be a device name. Any character not permitted in file names (by the **F\$PrsNam** system call) is taken to terminate the device name. Permitted characters are alphanumeric, '.', '_', and '\$'. The kernel does not interpret the remainder of the pathlist, or any of the pathlist if it does not begin with the '/' character.

Therefore elements of pathlists and the element separators can follow almost any convention, according to the specification prepared by the file manager writer. For compatibility with UNIX, however, elements are usually separated by the '/' character. RBF, Pipeman, and NFM use this convention. In addition, file managers usually use the **F\$PrsNam** system call to parse name elements.

Multiple pathlist elements usually refer to a directory hierarchy, but could be used for other purposes. In effect, the pathlist provides a mechanism for passing an ordered list of character string parameters to the file manager.

13.6 CREATE AND OPEN

For file managers without filing structure support (such as SCF) these calls are usually synonymous, and prepare for I/O on the device. Such file managers will normally give an error if the pathlist is not a simple device name. For file managers that do support a filing structure, Open should prepare for access to an existing file, while Create should create a new file (or give an error if a file of the same name already exists), and open it for access.

13.6.1 SCF

SCF treats Create and Open calls identically. SCF also "attaches" (**I\$Attach**) the "echo device" specified in the device descriptor (if one is given), and saves its device table entry address in the path descriptor field **PD_DV2**. The echo device is the device used for output when data is written to the device on which the path was opened (the "primary device"), and is usually the same as

the primary device – that is, keyboard input from a terminal on a serial port is echoed back to the terminal on the same serial port. SCF allocates a buffer of 512 bytes (256 bytes prior to OS-9 version 2.3) for input line editing of subsequent Read Line calls.

SCF calls the device driver's Set Status routine with the **SS_Open** function code. The call is made only for the primary device, even if the echo device is not the same as the primary device.

13.6.2 RBF

RBF parses the pathlist as described above, skipping the device name if the pathlist starts with a '/' character. If the pathlist starts with the '/' character and there are no following name elements, RBF opens the root directory of the device (a Create request with such a pathlist is returned a "file exists" error – **E\$CEF**). If there are following name elements, RBF looks in the root directory of the device for the first element. If the pathlist does not start with the '/' character, RBF looks for the first element in the current data or execution directory, depending on whether the "execute" bit (bit 2) of the requested modes byte is set. It then looks for the next element within that directory, and so on.

All elements in the pathlist other than the last must be directories, thus creating a tree structured directory system. In the Open call the last element may also be a directory, provided the "directory" bit (bit 7) of the modes parameter is set. The Create call cannot be used to create a directory – the Make Directory call must be used. The Create call fails with a "file exists" error – **E\$CEF** – if the last element already exists in the directory.

At each stage RBF checks that the file or directory permissions contain the requested mode bits, either in the public field of the permissions, or – if the caller is in the same group as the file creator (or is a super user) – in the private field of the permissions. If not, RBF returns a "file not accessible" error – **E\$FNA**.

In the Create call, RBF creates the file with the supplied permissions byte, which specifies read, write, and execute permissions for public and private access. Note that if the execute bit is set in the modes byte the pathlist is assumed to be relative to the current execution directory even if neither the public nor the private execute permission bit is set in the permissions byte. Conversely, the permissions byte may set public or private execute permission (or both) even if the modes byte does not have the execute bit set (causing the pathlist to be taken relative to the current data directory). If the

"initial size" bit (bit 5) is set in the modes byte, the file is created with the requested size, otherwise it is created empty (the size is zero).

Note that if an initial file size is requested, RBF will not create the file with more than one segment. If a segment as large as the requested size cannot be allocated, RBF allocates the largest segment it can. Therefore to guarantee that a file is created with the desired size, the Create call should be followed by a Set Status call with the **SS_Size** function code to set the file size.

When creating a file, RBF inserts a new entry in the parent directory, using the first available entry. That is, if a file has been deleted from the directory (the first byte of the file name field is zero), then RBF will use that entry for the new file. If all the entries in the directory are occupied, RBF extends the directory, as it would extend any file. Each directory entry is 32 bytes - 28 bytes for the name, followed by 4 bytes giving the Logical Sector Number (LSN) of the File Descriptor (FD) sector for the file (which RBF allocates and initializes when creating the file). Note that RBF only uses the last 3 bytes of the field, as RBF restricts LSNs to 24 bits. The name string is terminated by having bit 7 of the last character set. All unused bytes in the directory entry are set to zero.

RBF allocates a memory buffer equal to one sector in size for caching the file descriptor sector, and another sector buffer for managing read and write requests that start or end part of the way through a sector.

RBF calls the device driver Set Status routine with the **SS_Open** function code.

13.6.3 The File Descriptor Sector

Each file (including directories) has a File Descriptor sector, giving information about the file. The directory entry for the file contains the LSN of the File Descriptor sector. (The LSN of the File Descriptor sector of the root directory is in the **DD_DIR** field of LSN 0.) The File Descriptor sector contains the user ID and group of the creator (the "file owner"), the file permissions, the date the file was created, the date and time the file was last "modified" (opened for writing), and the file size.

The remainder of the sector contains the file segment list. Each entry in the list is 5 bytes, allowing up to 48 entries if the sector size is 256 bytes. (Prior to OS-9 version 2.4, a bug in RBF caused the last entry not to be used, so the file was limited to 47 segments.) The segment list describes the fragmentation of the file. The first entry points to the first part of the file,

the second entry points to the second part of the file, and so on, until the whole file has been located. The first 3 bytes of the entry give a 24-bit LSN, indicating the start of that fragment (segment). The other two bytes are a 16-bit number indicating the length of the segment – the number of contiguous sectors. The length of each segment will vary as required by RBF to build up a file containing the total number of sectors occupied by the file. In general, a file will be more fragmented if it has been extended since its initial creation, or if the disk free space is in many separate small fragments (as a result of many files being created and deleted). The File Descriptor structure is shown below:

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	FD_ATT	b	File attributes.
\$001	FD_OWN	b 2	File owner – group (first byte) and user ID.
\$003	FD_DAT	b 5	Date and time the file was last modified, as YYMMDDHHMM.
\$008	FD_LNK	b	Number of links to the file. This field is always one. RBF does not currently support file links, although the Delete routine decrements this field and deletes the file if the field is now zero.
\$009	FD_SIZ	1	File size (in bytes).
\$00D	FD_Creat	b 3	Date the file was created, as YYMMDD.
\$010	FD_SEG	...	The segment list.

Note that the file owner's group and user ID are held as byte values only, not as word values (as in the rest of the system). This is a historical legacy from the original OS-9 for the 6809, which did not have the concept of user groups. Therefore under OS-9/68000 the word field for the user ID was split into byte fields for group and user ID. When checking user and group numbers for file access, RBF uses only the low byte of the caller's group and user ID. For example, a user in group 256 will be able to access files belonging to group 0 (the super user group). For this reason it is not advisable to allocate group numbers and user IDs above 255.

The year numbers in the creation and modification fields are relative to 1900. Thus 1992 is expressed as 92 (\$5C). The **FD_Creat** field contains only the date of creation of the file, not the time. The **FD_DAT** field contains the date and time the file was last "modified" (opened with the write bit set in the modes byte). When the file is first created this field is set to the date and time of creation. Note that the time is held only to the minute. For this reason programs that perform an action dependent on the time a file was last updated can only resolve the update to the minute. For example, the **make**

utility may cause a file to be recompiled even though it has not been changed, because the previous compilation happened within a minute of the last edit of the source file, so the "last modified" times of the source and Relocatable Object Files are the same.

As the segment size is held in a 16-bit number, the segment length cannot exceed $64k-1$ (65535) sectors. This sets the first limit on the size of RBF files. For a sector size of 256 bytes, the file size is limited to $48 \times 65535 \times 256 = 768\text{Mbytes}$.

RBF always keeps the filing system up to date on the disk. In particular, any changes to the File Descriptor sector or the Allocation Bit Map (described below) are immediately written to disk. While this makes RBF a little slower than the disk file managers in some other operating systems, it ensures that the filing system is very robust. The computer can be turned off or reset while files are open, and the filing system will not be corrupted in any way. If this happens, files that were being extended (writing at the end of the file) may appear longer than expected. This is because RBF always extends a file by at least the "minimum segment allocation" size given in the **PD SAS** field of the device descriptor (to reduce file fragmentation), and then trims the file size back to the length actually used when the file is closed. Also, because RBF maintains a sector buffer for data being written to the file, the data most recently written to the file may not have been written to the disk.

13.6.4 The Allocation Bit Map

RBF manages the disk space using an Allocation Bit Map. This map uses a number of contiguous sectors, following the disk identification (ID) sector (LSN 0), and preceding the root directory. Each bit of each byte represents a "cluster" of sectors on the disk. Bit 7 of the first byte represents the first cluster (starting at LSN 0), and so on. A cluster is a group of contiguous sectors. All the clusters on a disk are the same size, but this may be different from the cluster size on another disk. The size of a cluster is set at format time, and recorded in the disk ID sector. It must be an integral power of 2 (1, 2, 4, 8 and so on). In this way RBF can manage a disk with a large number of sectors more quickly by grouping the sectors into clusters, so reducing the size of the bit map. If a bit is set in the bit map, it indicates that the cluster is in use (allocated to a file), or is defective (and so must not be allocated). The bit map is initially built by the **format** utility.

When RBF allocates sectors to a file it always allocates complete clusters. The first sector of the first cluster allocated to the file always contains the File Descriptor sector. Therefore if the cluster size is greater than 1, the first

entry in the segmentation list for the file will always start with the LSN immediately following the LSN of the File Descriptor itself.

When creating or extending a file, RBF determines the number of clusters needed. It then searches the bit map, looking for the first free block (bits are zero) large enough, and allocates the required number of clusters from the start of the block, by setting the bits in the bit map to ones. If there is no block large enough, RBF uses the largest free block that it found during the search as the first new segment of the file. If the file is now large enough to satisfy the caller's request, RBF terminates the allocation (even if the allocation is not as great as RBF had intended). Otherwise RBF searches again, allocating more clusters to more segments until enough space has been found in total (or the disk is full, in which case RBF de-allocates the sectors, and returns a "disk full" error – **ESFULL**).

RBF will not create a file segment whose clusters are not wholly represented within one bit map sector. This means that a segment cannot contain more clusters than 8 times the number of bytes in one sector. For 256 bytes per sector this is a limit of 2048 clusters per segment. This is the other limitation on file size under RBF. Therefore in the case of 256 byte sectors, if the cluster size is less than 32 the limit is the number of *clusters* per segment (2048), otherwise the limit is the number of *sectors* per segment (65535). For example, if the cluster size is 1, the maximum file size is $48 \times 2048 \times 256 = 24\text{Mbytes}$.

Note that these calculations assume that disk fragmentation does not further limit the size of each segment (for example, two files being written simultaneously will "leap frog" each other in the allocation bit map). Also, a sector size larger than 256 bytes greatly reduces the problem – the segment list for the file is larger (because the File Descriptor sector is larger), each bit map sector is larger, and the sectors allocated to the file are larger.

The cluster size is set by the '-c' option of the **format** utility. Because a small cluster size may restrict the file size on a large disk, while a large cluster size will waste space (disk space is allocated to files in whole clusters), it is reasonable to set a cluster size that allows a file to be as large as the whole disk, not allowing for disk fragmentation. For example, a cluster size of 4 would be suitable for a 100Mbyte disk with a sector size of 256 bytes (giving a maximum file size of 96Mbytes). Bearing in mind that the limitation imposed by the rule that a segment cannot contain more than 65535 sectors, the maximum file size for a disk with 256 byte sectors is 768Mbytes, so it would normally not help to set a cluster size greater than 32 with a sector size of 256 bytes.

If, however, the sector size is 512 bytes, the file descriptor segment list can hold 99 entries, and a bit map sector corresponds to 4096 clusters. The file size limit imposed by the segment restriction of 65535 sectors is $99 \times 65535 \times 512 = 3168\text{Mbytes}$, and the file size limit imposed by the bit map sector size for a cluster size of 1 is $99 \times 4096 \times 512 = 198\text{Mbytes}$. Thus a cluster size of 1 would be acceptable for disks up to 200Mbytes, and a cluster size of 16 is the maximum that need normally be used.

13.6.5 Access to the Whole Disk

RBF provides a "whole device" feature in the Open call. If the pathlist consists simply of a device name followed by the '@' character ("commercial at sign"), RBF opens the whole device as if it were a file. The program can then seek, read, and write to the disk without regard for the filing structure. The file pointer of the "file" starts (value 0) from the first byte of LSN zero. For example, a seek to location 1024 followed by a read of 256 bytes would read a sector - LSN 4 - of a disk with 256 byte sectors. Reading and writing can use any block size, just as for a normal file, and the "file size" is the size of the whole disk. Therefore all programs that do not use the file descriptor or directory information of a file can be used without modification. For example:

```
$ dump /d0@
$ merge /r0@ -b100 >/dd/ramdisk_image
```

Users other than the super user group (group 0) are only allowed read access, and are only allowed to read up to the end of the allocation bit map. This protects against damage to the filing system, and prevents unauthorized access to files.

13.7 CHANGE DIRECTORY

This call is normally only implemented by file managers that support a hierarchical directory structure, such as RBF. Other file managers will return an error - SCF returns **E\$BPNAM** ("bad path name").

The process descriptor has a 32 byte area (**P\$DIO**) for use by the kernel and file manager in remembering which are the current execution and data directories. The first 16 bytes are used for the data directory, and the remaining 16 bytes for the execution directory. The kernel uses the first long word of each half to store the device table entry address of the device with the current directory, and reserves the following word. The remaining 10

bytes of each half are available to the file manager. RBF stores the sector number of the file descriptor sector of the current directory.

As mentioned above, the kernel allocates and initializes a path descriptor (including making the **I\$Attach** system call) before calling the file manager. Effectively, the file manager is working with an open path. The kernel "closes" the path and de-allocates the path descriptor after calling the file manager. Note that the kernel does not call the Close routine of the file manager when doing this. Therefore the file manager must perform an implicit closure of the directory file before returning to the kernel.

After calling the file manager Change Directory routine, and providing no error was returned, the kernel increments the device use count in the device table. This is to "hold" the I/O sub-system in existence even though there may be no path open to the device. Note that the kernel does not decrement the use count of the device in the device table when the current directory is changed to another device, so the use count of a disk drive rises each time a "change directory" request is made on it. The **I\$Detach** system call (as made by the **deiniz** utility) must be used to decrement the use count to zero if it is necessary to remove the disk drive from the device table.

The calling program passes a "modes" byte, which must have either the read or the execute bit set, or both. If the read bit is set, the current data directory is changed to the requested pathlist. If the execute bit is set, the current execution directory is changed. Thus both the current data and execution directories can be changed (to the same directory) with one call, by setting both mode bits. This protocol is followed by the kernel, and must also be followed by the file manager.

RBF opens the directory in the normal way (so checking that the caller has permission to access the directory for read or execute as requested), and then saves the LSN of the directory's file descriptor sector in one or both of the entries in the path descriptor, depending on the mode bits set. If the write bit was set in the modes byte (as is done in the C library function **chdir()**), RBF also attempts to update the "last modified" date and time of the directory (in its file descriptor sector), but ignores any error (for example, if the disk is write protected).

13.8 MAKE DIRECTORY

This call is normally only implemented by file managers that support a hierarchical directory structure, such as RBF. Other file managers will return an error – SCF returns **E\$BPNAM** ("bad path name").

The kernel temporarily opens a path for the duration of this call, in a similar way to the Change Directory call. The kernel treats this call as if it were a Create call with the directory bit set in the permissions byte (which is otherwise illegal), followed by a Close call. The file manager is therefore called with a properly initialized path descriptor, and must perform an implicit close of the file it has just created before returning to the kernel. For its own purposes when "opening" the path, the kernel forces the permissions byte to be directory and write (\$82). In OS-9 version 2.3 and later, the kernel also clears all bits in the supplied modes byte other than the read and execute flags (bits 0 and 2), and bitwise ORs the resulting modes byte into the permissions byte, thereby ensuring that any bits set in the modes byte are also set in the permissions byte.

However, the kernel does not modify the caller's stack frame, so the file manager is passed the modes and permissions as specified by the calling program. As with the Create and Open calls, the kernel uses the execute bit of the modes byte to determine whether the directory is to be created relative to the current data or execution directory (unless the pathlist starts with a device name).

RBF uses the execute bit of the modes byte in the same way. The new directory is created with file attributes as specified by the permissions byte supplied by the calling program. The directory bit is also set in the file attributes. In addition to creating the directory (in the same way as any file), RBF writes two directory entries to the new directory - "parent" and "self". The "parent" entry is first, and has the name '..'. Its File Descriptor LSN field contains the LSN of the File Descriptor of the directory that contains this directory. The "self" entry has the name '.'. Its File Descriptor LSN field contains the LSN of the newly created directory itself. This information allows RBF to support pathlists that move up the directory hierarchy, such as '../DEFS'.

A directory is therefore created with an initial file size of 64 bytes. RBF ignores a request to create a directory with a larger size (indicated by setting the "initial file size" bit - bit 5 - of the modes byte). However, RBF allocates a first segment to the directory whose size is one sector less than the minimum segment allocation size, in anticipation of many small extensions to the directory (as files are created in the directory). Similarly, if the directory later requires additional disk space for a new entry, the new segment size is determined by the minimum segment allocation size (**PD_SAS** in the path descriptor) - the extra sectors above the actual directory file size are not de-allocated.

13.9 DELETE

This call is normally only implemented by file managers that support a filing structure, such as RBF. It could be used for other purposes – for example, to delete a window in a window management system. It is essentially the converse of the Create call. File managers that do not support this call will normally return an error – SCF returns **ESBMODE** ("bad mode").

The kernel's behaviour is almost identical to its Make Directory handler, except that it uses the modes byte without modification, and copies the modes byte to simulate a permissions byte for the purposes of initializing the path descriptor. Again, the execute bit of the modes byte is used to determine whether the pathlist is relative to the current execution or data directory. The kernel calls the Delete routine of the file manager, and then de-allocates the path descriptor. The file manager's Delete routine must locate and delete the file.

RBF finds the file in the same way as in the Open call, except that it also checks that the calling program has write permission for the file. RBF then trims the file size back to zero, in a similar way to the Set Status call "set file size". This de-allocates all of the clusters allocated to the file, except the first cluster, which contains the File Descriptor sector. Finally, RBF de-allocates this last cluster, and marks the file as deleted in the parent directory, by overwriting the first character of the name with a byte of zero.

Normally, if the file size is decreased, resulting in a segment table entry no longer being required, the "number of sectors" field of the entry is set to zero. From OS-9 version 2.3 onwards, RBF does not clear the segment list in the File Descriptor sector to zeros when trimming the file in the Delete routine. This allows a file that has just been deleted to be recovered, provided no files have been created or extended in the meantime. The only information not available is the first character of the file name in the directory entry. Such an "undelete" utility is not provided as standard with OS-9.

RBF will not allow a directory to be deleted. The "set attributes" Set Status call (**SS Attr**) must be used to remove the directory attribute of the file before deleting a directory file. RBF will only allow this to be done if all the entries in the directory have been deleted (other than the "self" and "parent" entries) – that is, the directory is empty. The **deldir** utility performs all three operations. It deletes all the files in a directory, removes the directory attribute from the directory, then deletes the file.

13.10 SEEK

This call requests a repositioning of the file pointer for subsequent reading or writing. It is therefore normally only implemented by file managers that support random access of data, such as RBF. It could be used for other purposes, such as setting the cursor position in a graphics display.

SCF does not support this call – it does nothing, but returns no error.

RBF allows any file pointer value to be set. The device is not accessed as part of the seek operation. If the file pointer is past the current end of the file, a subsequent Read or Read Line call will return an "end of file" error (E\$EOF), while a subsequent Write or Write Line call will cause the file to be extended. The file is extended to the length given by the file pointer set by the Seek call, plus the length of the Write or Write Line call. This leaves a portion of the file unwritten – it will contain whatever data was previously in the allocated sectors.

13.11 READ AND WRITE

These are the "raw" data transfer calls, to get data from or send data to a device. In the general philosophy of OS-9 I/O, these routines transfer the requested number of bytes unless an error occurs, or the end of the file is reached when reading. Other input termination conditions are file manager dependent. The file manager may also implement some processing of the data. However, the philosophy of these calls is in contrast to the Read Line and Write Line calls, so the data processing is usually minimal.

13.11.1 RBF

RBF performs no data processing. The number of characters transferred is always the number requested, unless a device driver error occurs, or the end of the file is reached in a Read call, or the disk (or the file's segment list) is full during a Write call. If a Read call attempts to read past the end of the file, no error is returned provided one or more bytes were read. The file pointer is moved to the end of the file. A subsequent Read call – reading starting at the end of the file – is returned an "end of file" error (E\$EOF). For both Read and Write calls, the file pointer is advanced by the number of bytes read or written.

When a Write call writes past the end of the file, RBF automatically extends the file to accommodate the new length. When extending a file, RBF checks whether the current last cluster in the file already has sufficient room to

extend the file. If not, RBF checks whether the last segment in the file has sufficient room (in case it was pre-extended by an earlier write). If not, RBF must allocate additional space for the file, as described above.

13.11.2 SCF

In the Read call, SCF echoes characters read (if echo is enabled in the path descriptor **PD_EKO** field) to the attached echo device (usually the same as the primary device). SCF terminates the call with an "end of file" error (**E\$EOF**) if the end-of-file character in the path descriptor (**PD_EOF**) is not zero, and matches the first character read. SCF also terminates the input prematurely if an input character is not zero, and matches the end-of-record character in the path descriptor options section (**PD_EOR**), or if the device driver returns an error.

In the Write call, SCF implements its "page pause" feature. That is, if the "page pause" flag is set in the path descriptor (**PD_PAU** is not zero), SCF will pause before sending a Carriage Return character if the total number of Carriage Return characters sent since SCF last read a character on this path is equal to the "number of lines per page" field (**PD_PAG**). Therefore "page pause" must be turned off in the path descriptor (**PD_PAU** set to zero) if binary data is to be sent that may include Carriage Return characters (byte value \$0D). Otherwise the output will be paused once the number of Carriage Return characters sent equals the "page length" field (**PD_PAG**), until a character is received from the device.

13.12 READ LINE AND WRITE LINE

The only strict difference in the OS-9 I/O philosophy in file manager operation between these routines and the Read and Write routines is that in addition to the reasons for termination of the Read and Write calls (given above), these routines terminate if a Carriage Return character is encountered. However, even this feature is a function of the file manager only. In addition, it is intended that file managers may implement more data editing in these calls than in the Read and Write calls.

Because the calls should terminate if a Carriage Return character is transferred, the actual number of characters transferred may be less than the number requested. Note that the Carriage Return character is transferred, and is included in the count of characters transferred. If no Carriage Return character is encountered, the transfer will terminate once

the requested number of characters has been transferred, just as with the Read and Write calls.

13.12.1 RBF

RBF behaves exactly as described above for the Read and Write calls (with no data processing), other than terminating the request when a Carriage Return character is read or written.

13.12.2 SCF

SCF behaves as described for the Read and Write calls, with some additional features. In the Write Line call, characters are converted to upper case if that option is enabled in the path descriptor (the **PD_UPC** field is not zero), and a Line Feed character (byte value \$0A) is automatically output after the Carriage Return character (if any) if the **PD_ALF** field is not zero. The SCF end of line pause is implemented. That is, if the device driver sets the "pause" flag in the Device Static storage because the "pause" character was received (non-zero, and matching the value in **PD_PSC**), SCF pauses output before outputting the Carriage Return character, until a character (other than the "pause" character) is received.

SCF also implements its end of page pause feature (as described above), and tab expansion – tab characters are expanded to spaces. That is, if a non-zero character matching the **PD_Tab** field of the path descriptor (usually \$09) is to be transmitted, SCF instead transmits space characters (byte value \$20) to bring the total number of characters sent since the last Carriage Return up to an integral multiple of the tab length field (**PD_Tabs**).

In the Read Line call SCF terminates when the character read matches the "end of record" character (**PD_EOR**), rather than the Carriage Return character. Normally the "end of record" character is set to be the same as the Carriage Return character. Note that input does *not* terminate when the requested number of characters has been input. SCF continues to input characters until the "end of record" character is received, discarding any characters that exceed the number requested. Input also terminates if the device driver returns an error, or if the "end of file" character (**PD_EOF**) is received and is the first character in the buffer. (Note that other characters can be input, provided they are deleted before the "end of file" character is entered.)

Note also that the "end of record" character is echoed without converting it to a Carriage Return character, and that the "automatic line feed" feature of

SCF is implemented only if a Carriage Return character is output, not an "end of record" character.

Input is through the SCF editing buffer, and so is restricted to 512 bytes (256 prior to OS-9 version 2.3), including the "end of record" character. The SCF input line editing features are operative during a Read Line call. The following list shows the line editing keys, giving the path descriptor field containing the key character value, and the standard setting for that field.

<u>PD field</u>	<u>Standard</u>	<u>Action</u>
PD_BSP	^H	Delete the character to the left of the cursor.
PD_DEL	^X	Delete all characters (delete line).
PD_RPR	^D	Redisplay all characters (reprint line). This is useful if the display has become corrupted, perhaps due to limitations of the display terminal.
PD_DUP	^A	Redisplay from the current position in the input buffer to the end of the line. This causes SCF to display characters from the input buffer, starting at the current cursor position, and stopping at the first "end of record" character encountered, or at the last character ever entered into the buffer. This allows an entered line to be repeated. It also allows a simple form of editing of a previous line, by typing in and so overwriting characters in the buffer, and then redisplaying the remainder of the buffer (and then perhaps backspacing over some characters, and overwriting those).

13.13 GET STATUS AND SET STATUS

These are "wild card" routines, and so their function is very much file manager dependent. Most file managers implement the **SS_Opt** function of the Set Status call, setting new path descriptor options. The file manager may restrict which fields of the path descriptor options section can be modified in this way.

File managers will normally pass all calls on to the driver, even if they have recognized and handled the call (unless the file manager itself generated an error). If the file manager has handled the call, and the driver returns an "unknown service request" error (**E\$UnkSvc**), the file manager should ignore the error. This permits drivers to choose to act on calls already handled by the file manager (such as a change of the path descriptor options section), or reject the call by returning the **E\$UnkSvc** error.

File managers may internally generate calls to the device driver. For example, by convention file managers make an **SS_Open** Set Status call

FILE MANAGERS

when opening or creating a file, and an **SS_Close** Set Status call when the last image of a path is closed. Note that if an internally generated call might also be made from a program, the file manager will need to pass any parameters by putting them in the caller's register stack frame (saving and afterwards restoring what was there before!), because the device driver cannot know whether the call was from a program, or was internally generated by the file manager.

SCF recognizes no calls other than the **SS_Opt** Set Status call. RBF implements several calls. They are described in detail in the OS-9 Technical Manual, and are also briefly listed below:

☐ Get Status

<u>Function</u>	<u>Description</u>
SS_Ready	Test for data available (always true).
SS_Size	Get file size.
SS_Pos	Get current file pointer.
SS_EOF	Test for file pointer at end of file.
SS_FD	Read part or all of the File Descriptor sector of the file.
SS_FDIInf	Read part or all of a File Descriptor sector, specifying the LSN of the File Descriptor sector (usually from reading the directory entry of a file). This permits a File Descriptor sector to be read without opening the file.

☐ Set Status

<u>Function</u>	<u>Description</u>
SS_Opt	Update the options section of the path descriptor from the caller's buffer.
SS_Size	Set a new size for the file - causes the file to be extended or truncated (trimmed).
SS_FD	Update the File Descriptor sector of the file from the caller's buffer. The segment list cannot be altered, and only two other fields can be altered: the owner's group and user ID (can only be altered by a super user), and the "last modified" date and time. Prior to OS-9 version 2.3, the date of creation could also be altered.
SS_Ticks	Set the maximum wait for a record to become unlocked. If the caller is subsequently put to sleep by RBF because it attempts to read a record on this path that is currently held by another process, RBF uses this as the parameter to the sleep call. A value of zero (the default) will cause an indefinite wait for the record to be released. A timeout while waiting causes RBF to return an E\$Lock error to the caller.

<u>Function</u>	<u>Description</u>
SS_Lock	Request RBF to lock part or all of a file. This will lock a record without the need for the calling process to read the record. A further call to this function will release any previously locked record. Hence a call with a record length of zero removes any lock held by this process on the file.
SS_Attr	Change the attributes (permissions) of the file. The calling process must be a member of the same group as the owner of the file, or be a super user. The directory bit of a directory file can only be cleared if the directory has no entries still in use (other than "self" and "parent"). The directory bit of the root directory cannot be cleared.

13.14 CLOSE

The Close routine is essentially the converse of the Open and Create routines. The file manager is requested to ensure that any resources allocated for the management of the path are de-allocated, and that all information about the file (if a filing system is supported) is up to date on the medium. However, the Close system call (**ISClose**) is also the converse of the Duplicate Path system call (**ISDup**). Therefore a Close request to the file manager may not cause the termination of a path, as there may be other duplications (or "images") still in existence.

Path images are counted in the path descriptor word field **PD_COUNT**. The kernel also maintains the byte field **PD CNT**, but this is for historical compatibility only. The file manager should terminate the path and de-allocate associated resources only when the Close routine is called with the **PD_COUNT** field at zero. When this occurs, a file manager will also typically generate a **SS_Close** Set Status call to the device driver.

13.15 CALLING THE DEVICE DRIVER

The file manager calls the device driver to carry out physical device operations, and to pass on Get Status and Set Status calls. The section on Device Drivers gives details of the device driver routines, and the calling conventions used by SCF and RBF. Note that the file manager must not call the Initialization and Termination routines of the device driver – these are called only by the kernel.

A device driver is a separate OS-9 memory module, and the file manager writer cannot know the address of the device driver at compile time. To call a device driver routine the file manager must calculate the routine address, using the device driver module address in the device table entry, and the

offset to the required routine (from the routine offset table in the device driver). The offsets within the table have been symbolically defined by Microware in the file 'DEFS/sysio.a'.

For example, to call the driver Read routine:

```
movea.l PD_DEV(a1),a0    get device table entry address
movea.l V$STAT(a0),a2    get device static storage
movea.l V$DRVR(a0),a0    get driver module address
move.l M$Exec(a0),d0     get offset to routine offset table
move.w D$READ(a0,d0.1),d0 get offset to read routine
jsr     0(a0,d0.w)       call the routine
```

The file manager will normally save the processor registers that it wishes to have preserved before calling the device driver. This avoids the need for the device driver writer to know which registers the file manager wishes preserved.

13.16 RESOURCE CONTROL

Paths and devices are system resources. Catastrophic results could occur if two processes were allowed concurrent access - system memory structures could be corrupted, and device operations confused. In general this cannot occur, because process rescheduling does not occur while a process is executing in system state, so the system call can finish its operations without worrying that it may be scheduled out, and another process scheduled in which will want to use the same resource.

However, it is possible for an operating system routine to explicitly go to sleep (**F\$Sleep** or **F\$Event**). This is normal practice in interrupt-driven device drivers. In this case another process may become the current process, and may attempt access to the same system resource.

The kernel controls concurrent accesses to the same path, using the **PD CPR** field of the path descriptor. If this field is non-zero when a process makes an I/O request on the path, the kernel assumes it to be the ID of a process currently executing an I/O operation on the path, and I/O queues the calling process on that process (**F\$IOQu** system call). Otherwise it puts the ID of the calling process in the **PD CPR** field, so holding the path for the calling process. On return from calling the file manager, the kernel releases the path by clearing the **PD CPR** field, and waking up the first process in the queue of processes queued on the calling process (if any).

However, the kernel does not implement any control of concurrent accesses to devices, or to "channels" within devices. This function is left for the file

manager or device driver. The file manager writer must decide what control of concurrent accesses the file manager will provide, and the device driver must implement any remaining functionality. SCF and RBF use a common mechanism, which removes the need for the device drivers to perform any control over concurrent accesses. They use the **V_BUSY** field of the device static storage to prevent concurrent accesses to the whole device, in the same way the kernel uses **PD_CPR** to prevent concurrent accesses to the path descriptor.

When the file manager wishes to acquire the device (for example, before calling the device driver), it checks the **V_BUSY** field. If it is zero, the device is free – the file manager copies the process ID of the calling process to this field, and makes its call to the device driver. On return, the file manager clears the **V_BUSY** field, and wakes up the first process (if any) queued on the current process. If **V_BUSY** is not zero, however, the file manager assumes it is the process ID of a process currently making a call into the device driver, and I/O queues the current process onto the process that is holding the device (**F\$IOQu** system call). This puts the current process to sleep. On wakeup, the file manager again tries to acquire the device, unless it decides that a fatal condition has occurred.

It is up to the file manager writer to decide when and how to acquire control of a device – it can be applied to the device as a whole, or to a "channel" on the device, or it can even be left entirely to the device driver. RBF and SCF behave somewhat differently. RBF acquires the device just before calling the driver, and releases it on return from the driver (unless it is manipulating the allocation bit map, in which case it hangs on to the device until it has finished the allocation bit map function). By contrast, SCF acquires the device (and the associated echo device, if any) at the start of the file manager routine, and does not release it (knowing that the kernel will perform an I/O unqueue operation when SCF returns to the kernel). This keeps text lines indivisible if multiple processes are writing to the same device.

Because only one field – **V_BUSY** – is used for device allocation, these file managers prevent concurrent read and write requests. This is why output cannot occur to the screen while a process is taking in input from the keyboard. This is not a fixed requirement, however. Provided the device driver can handle concurrent read and write operations, the file manager can allocate the device for read and write operations separately, using separate fields (defined by the file manager writer) in the device static storage. The same applies to the allocation of multiple channels within a device. In the most liberal case, the file manager will use separate allocation fields for read and write operations on each channel, leaving the device driver with the

FILE MANAGERS

responsibility to ensure that it does not cause conflicts of use of the device static storage, or of access to the physical device interface.

There is no simple system call to "I/O unqueue" a process waiting on the current process. The file manager must check the I/O queue fields of the current process's process descriptor, unlink the current process from the queue, and wake up the first process in the queue. If there were multiple processes in the original I/O queue, the remainder of the queue is now rooted in the process descriptor for the process that has been woken:

move.w	P\$I0QN(a4),d0	get ID of process queued on this
beq.s	Done	..none; no action
clr.w	P\$I0QN(a4)	clear the "next" link
moveq	#S\$Wake,d1	wake up the queued process
os9	F\$Send	by sending the wakeup signal
os9	F\$GProcP	get proc desc ptr of queued process
clr.w	P\$I0QP(a1)	clear the "previous" link

Done

However, as described for SCF above, the file manager does not need to perform an "I/O unqueue" operation when it has finished with the device or channel, because the kernel always performs such an operation after calling the file manager. Therefore the kernel will wake up the first process queued on the current process, whether the reason for queuing was because the process wanted to use the same path, or the same device. RBF performs an unqueue operation as soon as its call into the device driver has finished only because it aims to allow fair usage of the device by multiple processes concurrently.

Note that SCF does not use this device acquisition technique in a Get Status call. Therefore the **V_BUSY** field is not set, and if the device driver sleeps within the Get Status call, SCF may call any of the driver's routines as the result of another I/O call (from another process on another path to the same device).

13.17 A SKELETON FILE MANAGER

As with device drivers, part of the problem in writing a file manager is knowing where to start. This section shows a skeleton file manager in 68000 assembly language. The chapter on "Microware C and Assembly Language" shows how this can be adapted to form the core of a file manager written in C.

* Skeleton file manager

```

Typ_Lang    set      (FIMgr<<8)+Objct    module type and language
Att_Revs    set      ((ReEnt+SupStat)<<8)+0  module attributes and
*                                                    revision
Edition     set      1                    software edition number
            psect    skelmgr,Typ_Lang,Att_Revs,Edition,0,EntryTable
            use      /dd/DEFS/oskdefs.d

```

* Routine offset table:

```

EntryTable  dc.w      Create-EntryTable    create
            dc.w      Open-EntryTable      open
            dc.w      MakDir-EntryTable    make directory
            dc.w      ChgDir-EntryTable    change directory
            dc.w      Delete-EntryTable    delete
            dc.w      Seek-EntryTable      seek
            dc.w      Read-EntryTable      read
            dc.w      Write-EntryTable     write
            dc.w      ReadLn-EntryTable    read line
            dc.w      WriteLn-EntryTable   write line
            dc.w      GetStat-EntryTable   get status
            dc.w      SetStat-EntryTable   set status
            dc.w      Close-EntryTable     close

```

* Create

```

* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*

```

```

Create      movea.l  R$a0(a5),a0          get ptr to pathlist
            move.w   #E$BMode,d1         error - can't create
            ori      #Carry,ccr
            rts

```

* Open

```

* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor of current process
*           (a5) = caller's register stack frame
*           (a6) = System Globals
* Returns:  carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*

```

```

Open        movea.l  R$a0(a5),a0          get ptr to pathlist
            move.w   #E$BMode,d1         error - can't open
            ori      #Carry,ccr
            rts

```

FILE MANAGERS

```
* Make directory
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4, ccr (NOT a5/a6)
*
MakDir      movea.l  R$a0(a5),a0      get ptr to pathlist
            move.w   #$BMode,d1      error - can't make directory
            ori      #Carry,ccr
            rts

* Change directory
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4, ccr (NOT a5/a6)
*
ChgDir      movea.l  R$a0(a5),a0      get ptr to pathlist
            move.w   #$BMode,d1      error - can't change directory
            ori      #Carry,ccr
            rts

* Delete
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4, ccr (NOT a5/a6)
*
Delete      movea.l  R$a0(a5),a0      get ptr to pathlist
            move.w   #$BMode,d1      error - can't delete
            ori      #Carry,ccr
            rts

* Seek
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4, ccr NOT (a5/a6)
*
Seek        move.l   R$d1(a5),d0      get desired position
            rts                      no action - carry is clear
```

```

* Read
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
Read      move.l  R$d1(a5),d0      get (max) number of bytes to read
          movea.l R$a0(a5),a0      get ptr to buffer
          clr.l   R$d1(a5)         no bytes read
          rts                    no action - carry is clear

* Write
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
Write     move.l  R$d1(a5),d0      get (max) number of bytes to write
          movea.l R$a0(a5),a0      get ptr to buffer
          clr.l   R$d1(a5)         no bytes written
          rts                    no action - carry is clear

* Read line
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
ReadLn    move.l  R$d1(a5),d0      get (max) number of bytes to read
          movea.l R$a0(a5),a0      get ptr to buffer
          clr.l   R$d1(a5)         no bytes read
          rts                    no action - carry is clear

* Write line
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
WriteLn   move.l  R$d1(a5),d0      get (max) number of bytes to write
          movea.l R$a0(a5),a0      get ptr to buffer
          clr.l   R$d1(a5)         no bytes written
          rts                    no action - carry is clear

```

FILE MANAGERS

```
* Get status
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
GetStat      move.w    R$d1+2(a5),d0    get function code
              move.w    E$UnkSvc,d1      unknown function
              ori       #Carry,ccr       show error
              rts

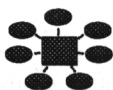
* Set status
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
SetStat      move.w    R$d1+2(a5),d0    get function code
              move.w    E$UnkSvc,d1      unknown function
              ori       #Carry,ccr       show error
              rts

* Close
* Passed:  (a1) = Path Descriptor
*          (a4) = Process Descriptor of current process
*          (a5) = caller's register stack frame
*          (a6) = System Globals
* Returns: carry set if error, with error code in d1.w
* May destroy: d0-d7/a0-a4,ccr (NOT a5/a6)
*
Close        tst.w     PD_COUNT(a1)      last duplication is closing?
              bne.s     Close10          ..no
              nop
Close10      rts                          no error - carry is clear

              ends                        end of code
```

CHAPTER 14

OS-9 INTERNAL STRUCTURE



OS-9 is a true multi-tasking operating system. Therefore it has many resource allocation and management jobs to perform. To keep track of resources, both allocated and free, it uses a number of different data structures in memory. Also, because OS-9 is aimed at a very wide range of applications, it has very few limitations on the number of each type of resource it can manage. Therefore, to avoid wasting memory, memory is dynamically allocated as required for these data structures.

The purpose of this chapter is to identify all of the important data structures used by the operating system, and to describe in detail all of the fields of the principal structures. Because these structures determine all of the resource control mechanisms of OS-9, an understanding of these structures gives a complete understanding of how the operating system allocates and controls all of the system resources.

The OS-9 kernel is written in assembly language. Therefore all of the data structures have been defined in assembly language files. Microware has used the technique of declaring all of the symbols in the structures as public symbols, assembling the files to ROFs, and merging them to the library 'LIB/sys.l'. The **make** file to do this is provided by Microware in 'DEFS/makefile'.

This library is used by the linker when creating the kernel. For this reason, these definitions files are absolutely definitive. They are not just descriptive, but have actually been used to create the kernel. Microware supplies all of these files with OS-9, in the 'DEFS' directory.

<u>File</u>	<u>Structures defined</u>
sysglob.a	System Globals data structure.
sysio.a	Device Table, Interrupt Polling Table, and Path Descriptor data structures.
iodev.a	Device Static Storage structure for the kernel.
scfdev.a	Device Static Storage structure for SCF
rbfdev.a	Device Static Storage structure for RBF
funcs.a	System call codes and error numbers.
io.a	Path Descriptor Options Section data structures.
module.a	Module header structures.
process.a	Process Descriptor data structure.

Most of the dynamically allocated tables (arrays) used by the kernel are dynamically extendible if the table becomes full. The kernel allocates a new table twice the size of the old table, copies the old table to the first half of the new table, and frees the old table's memory. This allows small tables to be allocated initially, to conserve memory on small systems.

Some tables are not dynamically extendible. The size of each of these tables is specified in the **init** configuration module, so the implementor can tailor the table size to the system requirements. All of these non-extendible tables relate to physical resources (such as devices) which cannot be dynamically created, so the required table size is known when the computer is designed.

14.1 THE SYSTEM GLOBALS

The operating system must have some way of finding the data structures that have been created. It does this through a root structure of which only one instance exists, known as the System Globals. This structure contains pointers to other structures, which in turn may contain pointers to other structures, and so on. It is impossible to know the addresses of any of the data structures in advance, as OS-9 imposes no constraints on the system memory map.

On coldstart there must be some mechanism, suitable for all systems, of locating memory where the System Globals structure can be built. There must also be a mechanism whereby the kernel can find the address of the System Globals whenever a system call is made, or an interrupt occurs. OS-9 uses a fixed feature of the 68000 family of processors - the Reset Stack

Pointer entry of the vector table. This is the first entry in the vector table, which is situated at address zero in all 68000 systems and most other systems. The higher members of the family have a Vector Base Register to point to the vector table. This is set to zero when the processor is reset, and most systems leave it unchanged. However, OS-9 for the 68020/030/040 does fetch the pointer relative to the Vector Base Register, so it is compatible with all configurations.

The Reset Stack Pointer is assumed to point to the middle of an 8k block of RAM. This is a reasonable constraint on the implementor, as the Reset Stack Pointer *must* point to some RAM, for the boot program to work. The upper half is used to store the System Globals. The lower half is the initial stack for the boot program. At the bottom of the 8k block is the OS-9 Exception Jump Table, described in the chapter on Exception Handling (about 2.5k bytes). Note that if the boot ROM uses the "CBOOT" code supplied by Microware, the total size of this block may exceed 8k (the top part, pointed to be the Reset Stack Pointer, is still 4k bytes).

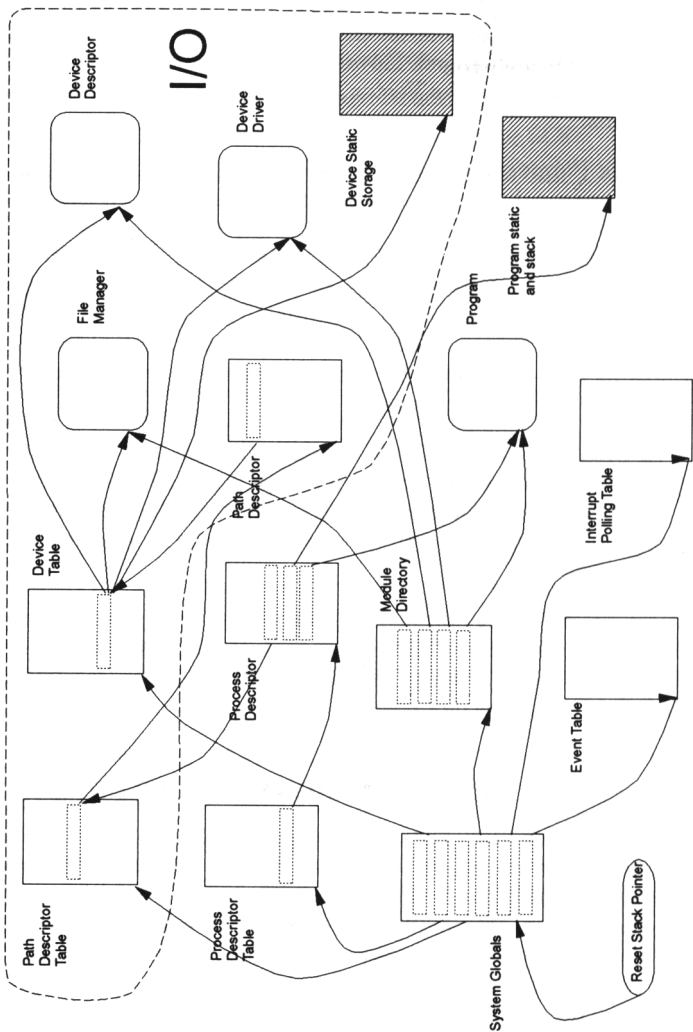
Although 4k bytes is reserved for the System Globals, the structure itself is not that big. This reservation allows for expansion of the System Globals structure without modification to the boot program. The remainder is used for the "system abort stack" (see the description of the **D_AbtStk** field in the System Globals below).

All operating system tables and memory structures are dynamically allocated from free memory, using the **F\$SRqMem** system call. Root pointers to tables and linked lists of memory structures are held in the System Globals. The System Globals structure is described in detail below.

14.2 THE OTHER SYSTEM MEMORY STRUCTURES

The following sections describe all of the principal memory structures used by the kernel. In the structure description tables, the "size" field shows the length of each item as 'l' for a long word, 'w' for a word, and 'b' for a byte. If more than one item is covered by one name – for example, an array – the size letter is followed by the number of items. Figure 7 on the next page shows the interconnections between the principal types of memory structure used by OS-9. The structures contain pointers giving the base addresses of other structures.

Even though not all types of structure are shown, the diagram is rather complex! Operating system structures are shown as ordinary rectangles, modules are shown as rectangles with rounded corners, and memory areas



• Figure 7 - OS-9 Memory Structures

whose structures are not completely defined by the kernel are shown as shaded rectangles. The top and right part of the diagram shows the structures used by the I/O system.

14.2.1 Process Descriptor

A process descriptor is allocated for each process. A process descriptor is 2k bytes in size. Approximately the first 1k bytes contain the process descriptor memory structure, for managing the process. The remainder is used for the stack during system calls made by that process. When the process dies the process descriptor memory is freed. The kernel keeps an array of addresses of process descriptors, known as the Process Descriptor Table. The process ID number is a direct index into this table.

The structure of the process descriptor is described in detail in the Process Descriptor section of this chapter.

14.2.2 Path Descriptor

A path descriptor is allocated for each path. Note that *duplications* of a path do not create a new path descriptor, only a new process local path number to the same descriptor. A path descriptor is 256 bytes in size, cleared to zeros when first allocated. The first 128 bytes are used for storing variables for managing the path. The first part of this structure is common to all paths, and is determined by the kernel. The remainder of the 128 bytes is for use by the file manager appropriate to the particular device, and its structure varies from one file manager to another. The other 128 bytes contain the "options section", initially copied from the options section of the device descriptor used to open the path. The structure of the options section is determined by the file manager.

The kernel keeps an array of addresses of path descriptors, known as the Path Descriptor Table. The "system path number" is a direct index into this table. Note that there is a unique system path number for each open path, but individual processes refer to their paths using "local path numbers", which have a range from 0 to 31. When a process opens or duplicates a path it is assigned the first free number in its range of local path numbers as a reference to the path. This local path number is translated by the kernel through a table in the process descriptor into the system path number that uniquely identifies the path. When a path is closed, the process's local path number is freed. When all duplications of the path have been closed, the path descriptor memory is freed.

OS-9 INTERNAL STRUCTURE

The definitions for the path descriptor shown below are taken from the file 'DEFS/sysio.a'.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	PD_PD	w	System path number of this path.
\$002	PD_MOD	b	Mode flags for the mode in which the file was opened (read, write, execute, directory, non-sharable).
\$003	PD_CNT	b	Number of local paths (duplications) open on this system path. This byte field is now redundant, having been replaced by the word field PD COUNT . However, the kernel still maintains it, for backward compatibility.
\$004	PD_DEV	l	Address of the device table entry for the device this path is open on. This field forms the link between a logical path and a physical device. The device table entry contains the addresses of the file manager module, device driver module, device descriptor module, and device static storage for the device.
\$008	PD_CPR	w	Process ID of the process currently making a system call on the path. The kernel sets this field at the start of a system call on this path, and clears it at the end of the system call. If this field is zero, there is no system call currently being executed on the path.
\$00A	PD_RGS	l	Address of the register stack frame of the process making a system call on this path. This address is used to read the parameters of the system call (they are passed in the caller's registers), and to return results, by modifying the register images in the stack frame.
\$00E	PD_BUF	l	Address of a data buffer. This field is not used by the kernel. It is for communication between the file manager and the device driver. For example, RBF puts the address of the buffer to read or write in this field before calling the device driver.
\$012	PD_USER	w 2	Group number and user ID of the process that opened the path.
\$016	PD_Paths	l	Address of the next path descriptor in the linked list of paths open on the same device. The device static storage contains the root pointer for this linked list.
\$01A	PD_COUNT	w	Number of local paths (duplications) open on this system path. This field is set to one by the kernel when the path is first opened, and incremented whenever a duplication of the path is made (another local path number is assigned to the same system path), either explicitly (by the ISDup system call) or implicitly (a child inheriting paths from the parent when forked). When a local path is closed this field is decremented. When it reaches zero

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			the path is actually closed, and the path descriptor memory is de-allocated.
\$01C	PD_LProc	w	Process ID of the last process to have made a system call on this path. This field is set at the same time as PD_CPR , but is not cleared when the system call finishes.
\$01E	PD_ErrNo	l	For the private use of the file manager - used to store the most recent error number during file manager operations (used for errno in file managers written in C).
\$022	PD_SysGlob	l	For the private use of the file manager - used by file managers written in C to save the address of the System Globals. The address of the System Globals is passed to the file manager by the kernel in the a6 register. File managers written in C normally put the address of the path descriptor in a6 , as this is the static storage address register assumed by the C compiler.
\$026		w 2	Reserved.
\$02A		b 86	For the private use of the file manager, as static storage associated with the path. The structure is defined by the file manager writer.
\$080	PD_OPT	b 128	Options section. The structure is defined by the file manager writer. The kernel initializes this area with a copy of the options section of the device descriptor the path was opened on.

14.2.3 Module Directory

The module directory is an array of structures containing the address, link count, header parity, and group identifier of each module present in memory. The structure of each entry is shown below. It is taken from the file 'DEFS/module.a'. Modules are described in the chapter on OS-9 Modules, Memory, and Processes.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	MD\$MPtr	l	Address of the module.
\$004	MD\$Group	l	Module group identifier - the address of the first module in the group.
\$008	MD\$Static	l	Size of the memory area allocated to contain the module group.
\$00C	MD\$Link	w	Link count of the module.
\$00E	MD\$MChk	w	Check word calculated from the module header bytes.

14.2.4 Device Table

The device table is an array of structures, one for each device currently active on the system (the device has had more "attaches" than "detaches"). A device is identified by its device descriptor module, so separate device table entries are created for different device descriptors, even if they refer to the same physical device. Each entry contains the addresses of the appropriate device descriptor, device driver, and file manager modules, plus the address of the device static storage, and a device use count. The entry is deleted (the use count, device descriptor address, and device static storage address are cleared to zeros) when the use count goes to zero.

Paths are linked to physical devices by means of a pointer (**PD_DEV**) in the path descriptor to the appropriate device table entry. The structure of a device table entry is defined in the file 'DEFS/sysio.a'.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	V\$DRIV	1	Address of the device driver module.
\$004	V\$STAT	1	Address of the device static storage.
\$008	V\$DESC	1	Address of the device descriptor module.
\$00C	V\$FMGR	1	Address of the file manager module.
\$010	V\$USRS	w	Current device use count.

The device table is *not* dynamically extendible. Its size is taken from the **init** module.

14.2.5 Device Static Storage

A device static storage is an area of memory used to control a single physical device interface. It may service multiple logical channels (for example, a floppy disk controller controlling four floppy disk drives). The size and usage of the device static storage is determined by the file manager and device driver controlling the device.

A separate device static storage is allocated to each device table entry unless both the device port address (in the device descriptor) and the device driver are the same as in an existing entry. In that case the kernel assumes that the new device descriptor is just another description of the same physical device (an "alias"), so the kernel uses the same device static storage as in the existing entry. The kernel clears a device static storage to zeros after allocation. The memory of a device static storage is freed when all device table entries referring to that device static storage have been deleted.

The first part of the device static storage is the same for all devices. The second part is defined by the file manager used to control the device. The third part is defined by the device driver used to control the device. Because the I/O system of OS-9 is tree structured (there is one kernel calling multiple file managers, and each file manager may call multiple device drivers), the size of the device static storage is finally determined by the linker when creating the device driver module, adding the universal definitions to the file manager definitions and the device driver definitions. The size of the device static storage required is therefore set in the device driver module header **M\$Mem** field by the linker.

Note that despite the similarity in names, the device static storage is very different from a process's static storage. It is a control structure associated with the device, rather than a private store of variables for a particular application. It does not have any space used for stack – the second half of the process descriptor of the calling process is used for the stack during a system call. System calls are effectively just system state subroutines executed by the calling process.

The following table describes the first (universal) part of the device static storage. The structure is defined in the file 'DEFS/iodev.a':

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	V_PORT	1	"Port address" – the address of the interface device. The kernel copies this field from the M\$Port field of the device descriptor module when it allocates the device static storage. The kernel makes no use of this field, and it does not interpret the port address except to decide whether to allocate a new device static storage, as described above. File managers also do not use this field – it is for the convenience of the device driver writer.
\$004	V_LPRC	w	The process ID of the last process to use the device. The kernel does not use this field. The SCF file manager sets it to the ID of the calling process on each I/O request. It may be used by SCF device drivers. For example, a serial port device driver's receive interrupt routine will typically send the "abort" signal to the last process to use the device when the "quit key" character is received.
\$006	V_BUSY	w	This field is not used by the kernel. It is typically used by the file manager to prevent concurrent calls into the device driver. The file manager checks this field before calling the device driver. If it is not zero, it is the process ID of the process currently making a call into the device driver, and the file manager "I/O queues" the current process on that process. Otherwise the file manager

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			copies the caller's process ID into this field and calls the device driver. When the device driver returns, the file manager clears this field.
\$008	V_WAKE	w	This field is for the private use of the device driver. It is usually used for communication between the interrupt handler of the device driver, and the main body of the device driver. If the device driver wants to wait for an interrupt, it copies the process ID of the calling process into this field and goes to sleep. On interrupt, the interrupt handler checks this field. If it is non-zero, the interrupt handler sends the wakeup signal to that process, and clears the field to zero (as a verifiable handshake to the main body of the device driver).
\$00A	V_Paths	1	Address of the first path descriptor in the linked list of path descriptors of paths open on this device (maintained by the kernel).
\$00E		1 8	Reserved.
\$02E	V_USER		The file manager part of the device static storage starts here.

14.2.6 Process Descriptor Table

This table is an array of the addresses of all existing process descriptors. The process ID is simply an index into this table. If an entry is zero, no process exists with that ID. Process IDs start with one (zero in a process ID field is used to mean "no process"), so the first location of the process descriptor table does not contain the address of a process descriptor. Instead, the first word contains the current size of the table (in terms of long word entries), and the second word contains the size of a process descriptor in bytes – 2048. The second entry (long word) of the table points to the process descriptor of the System Process, as this is always process 1.

14.2.7 Path Descriptor Table

This table is an array of the addresses of all existing path descriptors. A system path number is simply an index into this table. If an entry is zero, no path is open with that system path number. System path numbers start with one (zero in a system path number field is used to mean "no path"), so the first location of the path descriptor table does not contain the address of a path descriptor. Instead, the first word contains the current size of the table (in terms of long word entries), and the second word contains the size of a path descriptor in bytes – 256.

14.2.8 Interrupt Polling Table

This is an array of structures, one for each hardware interrupt handler routine currently installed. The System Globals contains 199 pointers, each of which is (if not zero) a root pointer to a linked list of these structures. Therefore there is one such linked list for each interrupt vector on which one or more interrupt handler routines has been installed. Interrupt handlers are installed using the **F\$IRQ** system call.

The interrupt polling table is *not* dynamically expandable – its size is set by an entry in the **init** module. Each entry is 18 bytes long, and has the following structure (defined in the file 'DEFS/sysio.a'):

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	Q\$LINK	1	Pointer to the next entry in the linked list for this vector number.
\$004	Q\$SERV	1	Address of interrupt handler routine.
\$008	Q\$STAT	1	Address of interrupt handler's static storage – normally the device static storage. This field is not checked or used by the kernel, except to identify the entry when it is to be deleted, and to pass the static storage address to the interrupt handler.
\$00C	Q\$POLL	1	Port address. This field is not checked or used by the kernel, except to pass the port address to the interrupt handler.
\$010	Q\$PRTY	b	Polling priority. An entry with a low polling priority number precedes an entry with a higher priority number, and so is called first to service an interrupt on that vector. A priority of zero means that the entry must be the only one on this vector.
\$011		b	Reserved.

14.2.9 Event Table

This is an array of structures, one for each event currently in existence. Events are created (and maintained) by the **F\$Event** system call. Each entry contains the event name, event number, the event's current value, and the link count of the event.

The event table is dynamically extendible. Its initial size is set by an entry in the **init** module – it is usually zero. Each entry is 32 bytes long, and has the following structure:

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Size</u>	<u>Description</u>
\$000	w	Event number.
\$002	b 12	Event name (null terminated if not 12 bytes long).
\$00E	l	Event value.
\$012	w	Wakeup increment.
\$014	w	Signal increment.
\$016	w	Event link count.
\$018	l	Address of process descriptor of first process waiting on the event – start of linked list of process descriptors.
\$01C	l	Address of process descriptor of last process waiting on the event.

The position within the table is called the event index. The first structure is index zero, the second is index one, and so on. The event ID that is returned by a call to create or link to an event is a long word – the high word is the event number, the low word is the event index. The kernel keeps a record of the last event number assigned, in the high word of the `D_EvID` field in the System Globals, initially zero. The kernel increments this word in the System Globals before using it to create the event ID for a new event.

The combination of event number and index as the event ID gives a high degree of confidence that a program cannot accidentally reference an event that has been deleted (the event ID will not match any existing event), while maintaining the speed of the event functions, which internally use the event index.

14.2.10 Service Dispatch Tables

The operating system calls are customizable – existing handler functions can be replaced by new handlers, and new system calls can be defined. This feature is provided through the dispatch tables, which contain the addresses of the system call handler functions. There are two tables – the system dispatch table, and the user dispatch table, for calls made from system state and user state respectively.

Each table consists of 512 long words. The first 256 long words are the addresses of the handler functions for each of 256 possible system call codes. The remaining 256 long words are the addresses of the (optional) private static storage for each of the handler functions. System call handler functions are installed by the `F$$$svc` system call, as described in the chapter on Exception Handling.

14.3 SYSTEM GLOBALS STRUCTURE

The System Globals structure is the root of all information in the system. Starting from this structure (whose address is held at memory location zero) all other memory structures can be located. The System Globals also contains system-wide variables, and system constants defined at coldstart. The System Globals structure is defined in the file 'DEFS/sysglob.a'. This section describes the function of each field (under OS-9 version 2.4).

Fields in the System Globals can be read directly by operating system components, or indirectly by user state programs using the **F\$SetSys** system call (or **_getsys()** C library function). User state programs can change certain locations, again by using the **F\$SetSys** system call, (or the **_setsys()** C library function). This technique must be used, rather than writing directly to the System Globals structure, as the kernel may need to take additional action when certain fields are altered. This is especially true for the fields that modify the behaviour of the process scheduler, as described in the chapter on Multi-tasking.

The table below shows the System Globals structure. The symbol names are defined in the file 'sysglob.a'. The named items are not necessarily contiguous. Microware has reserved several fields for future use, while some fields are skipped to keep word and long word fields on even addresses, and some fields are historical relics (from previous versions of OS-9) that are no longer used.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	D_ID	w	Set by the kernel to the module sync code \$4AFC after coldstart has finished. If set to \$6F6B (ASCII "ok"), the kernel does not check module CRCs during coldstart, effecting a "warm start".
\$002	D_NoSleep	w	Set non-zero to prevent the system process from sleeping.
\$020	D_Init	l	Set by the kernel to the address of the configuration module init , to speed access.
\$024	D_Clock	l	Set by the kernel to the address of its clock tick handler routine. The clock driver calls this routine every tick interrupt.
\$028	D_TckSec	w	Set by the clock driver to the number of ticks per second.
\$02A	D_Year	w	The current year (for example, 1993). This field and the following month and day fields are not dynamically maintained by the kernel. They are updated only when a

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			program requests the current date and time, using the F\$Time system call.
\$02C	D_Month	b	The current month (1 to 12).
\$02D	D_Day	b	The current day in the month (1 to 31).
\$02E	D_Compat	b	This is the first of two fields of bit flags used to modify the kernel behaviour to maintain the behaviour of previous versions of OS-9, or to cope with unusual hardware configurations (see below).
\$02F	D_68881	b	Floating Point Unit type (68020/030/040 systems only) 0 no FPU 1 68881 2 68882 40 68040
\$030	D_Julian	l	Julian day number - maintained by the kernel's tick handler routine. See the chapter on OS-9 System Calls for a full description of Julian dates.
\$034	D_Second	l	System time, as seconds left until midnight - maintained by the kernel's tick handler routine. Held in this unusual format to speed up the tick handler, which needs only to decrement this field and test for zero (new day).
\$03A	D_IRQFlag	b	Kernel flag - currently servicing an interrupt. This field is initialized to \$FF on coldstart. On entry to the kernel's interrupt handler the field is incremented - if it is now zero, the stack pointer register is set with the value in the D SysStk field, so switching to the "interrupt stack" - if an interrupt occurs unless an interrupt is already being serviced. On exit from the kernel's interrupt handler this field is decremented.
\$03B	D_UnkIRQ	b	The number of times an unknown interrupt has occurred in a row. If no interrupt handler acknowledges ownership of an incoming interrupt, the kernel increments this field. If an interrupt is handled successfully the kernel clears this field. If the field increments to zero (count of 256), the kernel masks interrupts to the level of the interrupt that cannot be handled. This mechanism is attempts to cope with hardware problems or hardware configuration errors.
\$03C	D_ModDir	l 2	Address of the module directory, and address of end of module directory memory plus one (to speed up module directory searches).
\$044	D_PrcDBT	l	Address of the Process Descriptor Table.
\$044	D_PthDBT	l	Address of the Path Descriptor Table.
\$04C	D_Proc	l	Address of the Process Descriptor of the Current

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			Process. The Current Process is the process currently executing. A system call routine will use this field to gain access to the process descriptor of the calling process. The Current Process is <i>not</i> in the Active Queue.
\$050	D_SysPrc	1	Address of the System Process Descriptor. The System Process (always process 1) is woken by the kernel's tick handler. Its function is to wake up processes in timed sleep, and to send alarms to processes.
\$054	D_Ticks	1	Ever incrementing tick count. Indicates the time since coldstart. Useful for timing intervals.
\$058	D_FProc	1	Address of the process descriptor of the process whose context is in the FPU registers. The kernel avoids unnecessary saving and reloading of the FPU registers (the FPU context can be very large) if other processes have not used the FPU (their FPU context shows "idle").
\$05C	D_AbtStk	1	System state abort stack pointer. This is intended to allow a graceful abort from a bus error within an interrupt handler. The abort stack is the remaining memory after the System Globals structure up to the 4k bytes reserved for the System Globals. On coldstart this field is initialized to point to the last long word of the abort stack (\$0FFC from the start of the System Globals). The address of the kernel's bus error handler function is written to that last long word of the abort stack. On entry to the kernel's interrupt handler the kernel decrements this field by 4 (allocating a long word on the abort stack), and writes its current (interrupt) stack pointer minus 4 to this long word pointed to by the abort stack pointer. The kernel then makes a subroutine call to the device driver's interrupt handler. Therefore during the interrupt handler of a device driver this field points to a recovery stack pointer value which can be loaded into the system stack pointer, after which an <i>rts</i> instruction will return to the kernel. On exit from the kernel's interrupt handler the kernel increments this field by 4, so ditching the saved interrupt stack pointer. Apart from maintaining this field as described, the kernel does not use it itself, even within its bus error handler.
\$060	D_SysStk	1	Address of the stack memory to use during interrupts. During the kernel coldstart this field is initialized to the address of the System Globals, so the interrupt stack is the memory below the System Globals. Once the kernel has linked to the <i>init</i> module it uses the "size of interrupt stack" field to allocate a separate area of memory, whose top address plus one is placed in this

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			field (push down stack). The default value for the size of this stack when creating the <code>init</code> module is 1k bytes.
\$064	D_SysROM	1	Boot ROM execution entry point - the address of a branch table in the boot ROM. This gives access to the non-interrupt driven input/output functions for the system console, which may be useful for announcing errors from within interrupt handlers.
\$068	D_ExcJmp	1	Address of the Exception Jump Table.
\$06C	D_TotRAM	1	Total RAM found by the boot program.
\$070	D_MinBlk	1	Process minimum allocatable block size - the minimum size of memory that can be allocated and freed by the software memory management functions. Memory is always allocated in multiples of this value (currently 16).
\$07C	D_BlkSiz	1	System minimum allocatable block size - the minimum size of memory that can be managed by the Memory Management Unit, if present, otherwise a default value (currently 256).
\$080	D_DevTbl	1	Address of the Device Table.
\$088	D_AutIRQ2	1 7	68070 on-chip I/O autovector interrupt polling table root pointers. The 68070 processor has on-chip interrupt sources not present in other members of the 68000 family.
\$0A4	D_VctIRQ	1 192	Vectored interrupt polling table root pointers. The 68000 family supports vector numbers 64 to 255. The kernel subtracts 64 from the vector number to form an index into this table.
\$3A4	D_SysDis	1	Address of the system state service dispatch table.
\$3A8	D_UsrDis	1	Address of the user state service dispatch table.
\$3AC	D_ActivQ	1 2	Active process queue pointers. The address of the first and last process descriptors in the linked list of process descriptors of active processes. These are processes that are requesting processor time (but excluding the current process - the process currently executing).
\$3B4	D_SleepQ	1 2	Sleeping process queue pointers - the linked list of processes in timed and untimed sleep. This list is ordered by time before wakeup, with the untimed sleeping processes at the end of the list.
\$3BC	D_WaitQ	1 2	Waiting process queue pointers - the linked list of processes waiting for a child process to die.
\$3C4	D_ActAge	1	Active queue age. This is the decrementing "system age" used in the management of process scheduling.
\$3C8	D_MPUTyp	1	Microprocessor in use, as an integer

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			(68000, 68010, 68020, 68030, 68040, 68070, 68300).
\$3CC	D_EvTbl	1 2	Address of event table and address of end of event table memory plus one (to speed up event table searches).
\$3D4	D_EvID	1	Last event number used. Only the high word of this field is used. Initially set to zero, this word is incremented before an event is created. It is used as the high word of the event ID of the event created (the low word of the event ID is the index into the event table for the event structure used for the new event).
\$3D8	D_SPUMem	1	Address of the static storage for the System Security Module. If this field is zero, inter-task memory protection is not in use (SSM is not active).
\$3DC	D_AddrLim	1	Highest memory address found during startup (both RAM and ROM).
\$3E0	D_Compat2	b	This is the second of two fields of bit flags used to modify the kernel behaviour to maintain the behaviour of previous versions of OS-9, or to cope with unusual hardware configurations (see below).
\$3E1	D_SnoopD	b	The kernel sets this field non-zero if all processor memory data caches are coherent ("snoopy") or no data caches exist. That is, the caches do not need flushing after another bus master (for example, a DMA controller) has written to memory. The internal caches of a 68040 are normally snoopy. Other caches are usually not snoopy.
\$3E2	D_ProcSz	w	The size of a process descriptor (currently 2k bytes). This value should be used to calculate the size of the system state stack, if system state stack checking code is being written.
\$3E4	D_PollTbl	1 8	Autovector interrupt polling table root pointers. The 68000 family supports 7 autovectors (one for each interrupt level). The autovector number is 24 plus the interrupt level of the incoming interrupt, 1 to 7. The kernel subtracts 24 from the vector number to generate an index into this table. The first location is therefore never used (OS-9 does not provide support for the "Spurious Interrupt" exception).
\$404	D_FreeMem	1 2	Address of the first and last entries in the linked list of memory area descriptors in the memory colour node table. As there is no system call to dynamically introduce new memory areas to the system, the colour node table (built at coldstart) is never changed.

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
The next three fields are not currently used. They were intended for use in a multi-processor version of OS-9:			
\$40C	D_IPID	w	Inter-processor identification number.
\$410	D_CPUs	1	Address of the array of processor descriptor list heads.
\$414	D_IPCmd	1 2	Start and end addresses of the inter-processor command queue.
The next three fields are used only by the F\$CCtl system call installed by the syscache kernel customization module (described in the chapter on OS-9 System Calls):			
\$764	D_CachMode	1	68020/68030/68040 Cache Control Register current setting in abstracted form.
\$768	D_DisInst	1	Instruction cache disable request depth - permits nested calls to disable the instruction caches. If non-zero, the instruction caches are currently disabled.
\$76C	D_DisData	1	Data cache disable request depth - permits nested calls to disable the data caches. If non-zero, the data caches are currently disabled.
\$770	D_ClkMem	1	Address of a clock tick thief's static storage. By substituting the address of its own tick handler in the field D_Clock , an operating system component can be called on every tick interrupt. Its handler must finish with a jump to the kernel's tick handler, whose address was saved from the original value in D_Clock . This requires that the handler have some static storage. The address of that static storage can be written here. System state alarms make this technique redundant for almost all purposes.
\$774	D_Tick	w	Number of ticks remaining in the current second. This field is initialized by the kernel from the field D_TckSec , and is decremented by the kernel's tick handler. When it reaches zero the tick handler re-initializes the field, and decrements the field D_Second .
\$776	D_TSlice	w	Ticks per time slice (copied by the kernel from the init configuration module).
\$778	D_Slice	w	Number of ticks remaining in the current time slice. The kernel initializes this field from D_TSlice when it starts a time slice for a process. The kernel's tick handler decrements this field. When the field reaches zero, the tick handler sets the "timed out" flag in the process descriptor of the current process (see D_Proc).
\$77C	D_Elapse	1	Number of ticks remaining before the system process should be woken. When a process requests a timed sleep

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			or an alarm signal, the kernel determines whether it will be the first process to need a wakeup or a signal. If so, it sets this field to the number of ticks to elapse before the system process needs to take action. The kernel's tick handler decrements this field (if it is not zero), and wakes up the system process when it reaches zero. The system process performs the necessary wakeup(s) and sends the necessary signal(s), and then recalculates the time to the next wakeup or alarm, setting this field accordingly (or to zero if none).
\$780	D_Thread	1 2	Addresses of the first and last thread queue linked list entries in the thread list, for the list of immediate or absolute time alarm signals. The kernel makes an entry in this list (allocating a thread structure from system memory) when a process requests an alarm signal by an absolute time, or which the kernel calculates requires an immediate signal. The entries are linked, ordered by the time of the alarm.
\$788	D_AlarTh	1 2	Addresses of the first and last thread queue linked list entries in the thread list, for the list of relative time and cyclic alarm signals. The kernel makes an entry in this list (allocating a thread structure from system memory) when a process requests an alarm signal by a relative time, or at repeating intervals. The entries are linked, ordered by the time of the alarm.
\$790	D_SStkLm	1	Interrupt stack memory base address.
\$794	D_Forks	1	Number of processes in existence.
\$798	D_BootRAM	1	Total amount of RAM found by the boot program.
\$79C	D_FPUSize	1	Maximum size of FPU saved state frame.
\$7A0	D_FPUMem	1	Address of static storage for the FPU emulator (for 68040).
\$7A4	D_IOGlob	b 256	This area of memory is for use by implementation-dependent operating system components, such as device drivers. Microware have reserved the first 32 bytes. This memory should only be used for variables that can only occur once in the system, such as images of processor board write-only registers. The structure of this memory is defined in 'DEFS/ioglob.a', which the implementor should edit as necessary (and then remake the 'LIB/sys.l' library).
The following three locations are used to modify the behaviour of the kernel's process scheduler (see the chapter on Multi-tasking). They can be altered by means of the F\$SetSys system call (and must not be modified directly):			
\$8A6	D_MinPty	w	Minimum process priority - processes with a lower

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			priority receive no processor time.
\$8A8	D_MaxAge	w	Maximum process priority – processes with a higher priority execute strictly in priority order. Processes with a lower priority are scheduled normally, but receive no time if a high priority process is active.
\$8AA	D_Sieze	w	(sic) If this field is not zero it is assumed to be a process ID. The kernel will not give time to any other process until this field is changed.
\$8AC	D_Cigar	l	The bytes of this field contain version numbers for the more important of the memory structures:
\$8AC		b	System Globals version – currently 1
\$8AD		b	Process descriptor version – currently 1
\$8AE		b	Module directory entry version – currently 1
\$8AF		b	Module header version – currently 1
\$8EC	D_SysDbg	l	Address of the entry point of the ROM based debugger. A user program (such as the 'break' utility) can call the ROM based debugger using the F\$SysDbg system call. This field is initialized during the kernel coldstart to the "boot entry point" address passed by the boot program, plus 16.
\$8F0	D_DbgMem	l	The System State Debugger program uses this field to save the address of its static storage. A non-zero value indicates that the debugger is active.
\$8F8	D_Cache	l	Address of the RBF disk cache buffer header. A non-zero value indicates that disk caching is active.

The **D_Compat** and **D_Compat2** fields contain bit flags to modify the behaviour of the kernel, to retain compatibility with earlier versions of OS-9, and to cope with hardware peculiarities. The bit flags of **D_Compat** are:

<u>Bit</u>	<u>Meaning when set</u>
0	Save all processor registers on interrupt (instead of just a subset).
1	Don't use the 68000 stop instruction to wait for interrupt when no processes are active – loop in software.
2	Don't implement the "sticky modules" feature.
3	Don't enable 68030 cache burst mode (used by the syscache kernel customization module).
4	Fill memory with a pattern when allocated or freed.

- 5 Don't attempt to start the clock driver during kernel coldstart (otherwise the kernel executes the **F\$STime** system call with a date of zero, to start the clock ticks and read a battery-backed time-of-day clock if one exists).

The bit flags of **D_Compact2** are:

<u>Bit</u>	<u>Meaning when set</u>
0	External instruction cache is snoopy or non-existent.
1	External data cache is snoopy or non-existent.
2	Processor instruction cache is snoopy or non-existent.
3	Processor data cache is snoopy or non-existent.
7	Kernel should not disable data caching during I/O system calls.

A snoopy (or coherent) memory cache is one that watches bus activity while another bus master is active. It automatically updates (or invalidates) its contents if the bus master writes to a memory location that is also held in the cache, and automatically inhibits the memory and supplies the cached value if the bus master tries to read a memory location that is "stale" because a "dirty" location in the cache has not yet been flushed to memory.

If the data caches are not snoopy (or non-existent), then device drivers that control devices with DMA must flush the caches before writing to the device or after reading from the device.

14.4 PROCESS DESCRIPTOR STRUCTURE

The Process Descriptor contains all the variables needed to control a process and record its resource allocations. It also contains the memory to be used as stack space during system calls made by the process. This removes the need for the program to reserve space in its stack for use by the operating system. The Process Descriptor structure is defined in the file 'DEFS/process.a'. This section describes the function of each field (under OS-9 version 2.4).

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	P\$ID	w	Process ID of this process.
\$002	P\$PID	w	Process ID of parent process. This field is zero if the parent has died.

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$004	P\$SID	w	Process ID of next sibling process – forms a chain of processes that are children of the same parent. This field is zero if this process is the last in the chain for this parent (or it is the only child of the parent).
\$006	P\$CID	w	Process ID of the first child process. The P\$SID field of the first child process holds the process ID of the second child process, and so on. In this way the kernel keeps track of all the children of a process. This field is zero if the process has no children.
\$008	P\$sp	l	System stack pointer, saved on the last system call made from user state, after the processor registers have been stacked. During a system call this field points to a stack frame of the processor registers of the calling user state program, while the a5 register points to the stack frame of the caller, whether system or user state.
\$00C	P\$usp	l	User stack pointer, saved on the last system call made from user state. During a system call this field points to the stack of the calling user state program.
\$010	P\$MemSiz	l	Not used.
\$014	P\$User	w 2	User group number and user ID of the user who forked the process (may be changed using the F\$SUser system call).
\$018	P\$Prior	w	Process priority.
\$01A	P\$Age	w	Process "age". This field is not used by the kernel, but is calculated when a copy of the process descriptor is requested using the F\$GPrDsc system call. If the process is not active it is set equal to the process's priority. Otherwise it is set equal to the process's scheduling constant minus the current system age (D_ActAge), which is an indication of the number of processes that have been put into the active queue (including rescheduling the current process when its time slice expires) since the process itself was put in the active queue. If the calculated value exceeds 10000 it is assumed to be "unreasonable", and a value of -1 (\$FFFF) is set. If the "maximum age" field in the System Globals D_MaxAge is not zero, and the calculated age of this process is greater than or equal to the "maximum age", the age is set to the maximum age minus one. None of this affects the scheduling of the process.
\$01C	P\$State	w	Only the high byte of this field is used. This byte is a set of bit flags indicating the current state of the process. The flags are described below.
\$01E	P\$Task	w	Process "task number" – for use by an SSM controlling an MMU that can store multiple memory maps at the

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			same time.
\$020	P\$QueueID	b	This field contains an ASCII printable character, indicating which queue (linked list) the process descriptor is currently in. The possible values are listed below.
\$021	P\$SCall	b	The function code of the last system call executed in user state.
\$022	P\$Baked	b	A kernel check flag - non-zero if the process was created by the F\$Fork system call.
\$024	P\$DeadLk	w	ID of the process to which an I/O deadlock has been lost.
\$026	P\$Signal	w	Signal code of pending signal. This is the code of the most recently received signal. (Note: prior to OS-9 version 2.4, this was the <i>oldest</i> signal, that is, the signal to be handled first.) This field is zero if there is no signal pending.
\$028	P\$SigVec	l	Address of process's signal handler function. If this field is zero, the process has not installed a signal handler function - the kernel will kill the process if a signal is received.
\$02C	P\$SigDat	l	Address of the data space of the signal handler function - normally the process's static storage.
\$030	P\$QueueN	l	Address of the process descriptor of the next process in the queue (linked list). This field and the following field provide the links for a doubly linked list of process descriptors. Depending on the value in P\$QueueID, this will be the active queue, the sleeping queue, the waiting queue, or an event queue.
\$034	P\$QueueP	l	Address of the process descriptor of the previous process in the queue.
\$038	P\$PModul	l	Address of the program module that was forked for this process.
\$03C	P\$Except	l 10	Addresses of the process's handler functions for the "hardware exceptions" (bus error, address error, illegal instruction, and so on). If a field is zero, the process has not installed a handler for the corresponding exception. In that case, the kernel will kill the process if that exception occurs while the process is executing in user state. See the chapter on Exception Handling.
\$064	P\$ExStk	l 10	Addresses of the static storage areas in which to build a stack frame of the processor registers if a "hardware exception" occurs. If a field is zero, the kernel will build the stack frame on the process's user state stack when the corresponding exception occurs.

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$08C	P\$Traps	1 15	Addresses of the trap handler modules installed to handle trap #n instructions 1 to 15. If a field is zero, the process has not installed a trap handler module (using the system call F\$TLink) for the corresponding trap #n instruction.
\$0C8	P\$TrpMem	1 15	Addresses of the static storage memory areas allocated for the trap handler modules. If a field is zero, the trap handler's module header shows the trap handler needs no static storage, or no handler is installed for that trap #n instruction.
\$104	P\$TrpSiz	1 15	Sizes of the static storage areas allocated for the trap handler modules.
\$140	P\$ExcpSP	1	System state recovery stack pointer - value to place in the system stack pointer if a "hardware exception" occurs while this process is executing in system state. Note: if a hardware exception occurs while an interrupt is being serviced, the kernel regards this as a fatal error, and reboots the system.
\$144	P\$ExcpPC	1	Address of the system state hardware exception handler. The kernel causes execution to divert to this address if a "hardware exception" occurs while the process is executing in system state. If a hardware exception occurs while the process is executing in system state and this location is zero, the kernel regards this as a fatal error, and reboots the system. At the start of a system call the kernel initializes this field and P\$ExcpSP to values that simply cause an early termination of the system call, with an abort of the process. At the end of a system call the kernel restores the previous values (usually zero). An operating system component (such as a device driver) can substitute its own values during a call, in order to take more appropriate action on exception.
\$148	P\$DIO	b 32	An area in which to store information about the current data and execution directories. The first 16 bytes are for information about the current data directory. The other 16 bytes are for information about the current execution directory. In each area the kernel uses the first 4 bytes to store the address of the device table entry for the device on which the directory resides, and reserves the following two bytes. The remaining 10 bytes are for use by the file manager controlling the device, as shown in the file 'DEFS/sysio.a'.
\$168	P\$Path	w 32	Table of system path numbers for open paths. When the process opens (or inherits) a path it is given a local path number in the range 0 to 31. The local path number is

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			an index into this table, giving the system path number for the path, which identifies the appropriate path descriptor. If a field is zero, the process does not have a path open with that local path number.
\$1A8	P\$MemImg	1 32	Table of addresses of memory areas allocated by the process, including the static storage ("primary data area"), and memory allocated by user state trap handlers (excluding the static storage of the trap handlers). When the process makes a memory allocation request, the kernel saves the address of the allocated memory in this table. The process can have up to 32 outstanding memory allocations at any one time. If a table entry is zero, there is no memory allocation corresponding to that table entry. The table is always ordered by memory address (low addresses first). If an entry is erased (because the memory is returned), the entries above are moved down, so the table has no "holes". If two or more allocated areas are contiguous, they are coalesced into one table entry.
\$228	P\$BlkSiz	1 32	Table of sizes of allocated memory areas. The size is the actual size allocated from the system free memory pool as the result of a memory request, and so is always a multiple of the system minimum allocatable block size. The area may contain free fragments that have not yet been given to the program, which is able to allocate memory in multiples of the process minimum allocatable block size (which is smaller than the system minimum allocatable block size). The kernel keeps track of these fragments - which belong to the process, but have not yet been allocated to a program request - through the linked list rooted in P\$frag .
The next three fields are used only if the process was created by the F\$DFork system call (a debugged process). The process is then under the control of its parent (the debugger):			
\$2A8	P\$DbgReg	1	Address of the register stack frame buffer in the parent's static storage. The kernel copies the process's registers to this stack frame after an F\$DFork or F\$DExec system call, and sets the process's registers from this stack frame when an F\$DExec system call is made by the parent.
\$2AC	P\$DbgPar	1	Address of the parent's process descriptor. If this field is zero, the process is not a "debugged" process.
\$2B0	P\$DbgIns	1	Total number of instructions executed in user state so far within an F\$DExec system call from the parent. This field is not used if the F\$DExec call specified "full speed" execution.

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$2B4	P\$UTicks	1	Number of tick interrupts that have occurred while this process was the current process in user state.
\$2B8	P\$STicks	1	Number of tick interrupts that have occurred while this process was the current process in system state.
\$2BC	P\$DatBeg	1	Date (in Julian form) when the process was forked.
\$2C0	P\$TimBeg	1	Time (seconds since midnight) when the process was forked.
\$2C4	P\$FCalls	1	Number of system calls (other than I/O system calls) made in user state.
\$2C8	P\$ICalls	1	Number of I/O system calls made in user state.
\$2CC	P\$RBytes	1	Number of bytes read (I\$Read and I\$ReadLn) in user state without error.
\$2D0	P\$WBytes	1	Number of bytes written (I\$Write and I\$WritLn) in user state without error.

The next two fields are used in building a queue of processes waiting for an I/O resource (such as a path or device). The kernel or file manager, on finding that a requested I/O resource is already in use by another process, will add this process to the I/O queue on that process using the **F\$IOQu** system call. When the kernel is finishing an I/O system call, it checks the process descriptor to see if it has an I/O queue (**P\$IOQN** is not zero). If so, it wakes up the first process in the queue (the process ID in **P\$IOQN**).

The queue is ordered by the scheduling constants of the processes at the time they were put in the queue. A process being put in the queue is inserted after other processes with the same or greater scheduling constants (see the chapter on Multi-tasking).

\$2D4	P\$IOQP	w	The process ID of the previous process in the I/O queue this process is waiting in. If this field is zero, this process is not I/O queued.
\$2D6	P\$IOQN	w	The process ID of the next process in the I/O queue. If the P\$IOQP field is zero, this process has the I/O resource, and this field is the process ID of the process that will be woken when the resource becomes free.
\$2D8	P\$Frag	1 2	<i>Not used (historical, from before coloured memory).</i>
\$2E0	P\$Sched	1	Scheduling "constant". This field is calculated when the process is put into the active queue. It is the sum of the system age (D_ActAge) at that time and the process's priority. It determines the position of the process in the active queue. See the chapter on Multi-tasking for a full description.
\$2E4	P\$SPUMem	1	Address of memory allocated by the System Security Module to contain the process's memory map as required for the MMU. This field is zero if the SSM is not used.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
The next two fields are set by the kernel as part of the F\$DExec system call made by the debugger to request that the process execute one or more instructions. These fields are not used if the process is not a "debugged" process (created by F\$DFork).			
\$2E8	P\$BkPtCnt	1	Number of breakpoints set. The F\$DExec system call specifies a number of breakpoints for the kernel to set in the program being debugged.
\$2EC	P\$BkPts	w 16	If "full speed" execution was requested by an F\$DExec system call, the breakpoints are set by the kernel by writing an illegal instruction at each breakpoint location. The instruction word that was at each breakpoint location is saved in this table by the kernel, and restored when the program halts (breakpoint, exit, or hardware exception). Otherwise (trace mode execution) the kernel sets the process into trace mode, so the process halts after each instruction with a trace exception. The kernel then compares the process's program counter with each breakpoint address in the list specified in the F\$DExec call.
\$30C	P\$Acct	1 8	This space is available for use by a "user accounting module". A user accounting module is a kernel customization module that is called whenever a process is forked or terminated, to keep track of the use of resources by users.
\$32C	P\$Data	1	Address of the process's static storage ("primary data area") allocated by the kernel when the process is forked. Also the first entry in the P\$MemIn table. It is this memory area that is expanded (or contracted) by the F\$Mem system call.
\$330	P\$DataSz	1	Size of the process's static storage, including the stack, but excluding the parameter string.
\$334	P\$FPUsave	1	Address of the memory area (allocated when the process is forked) in which to save the register frame and context of the FPU when the process ceases to be the current process. If the system does not have an FPU, this field is zero.
The next two fields have a similar purpose to the fields P\$Except and P\$ExStk . They are used for the additional "hardware exceptions" that can be generated by the FPU. If the system does not have an FPU, these fields are not used.			
\$338	P\$FPEcpt	1 7	Addresses of the process's handler functions for the "FPU exceptions" (divide by zero, not a number, and so on).
\$354	P\$FPExStk	1 7	Addresses of the static storage areas in which to build a stack frame of the processor registers if an "FPU exception" occurs. If a field is zero, the kernel will build the stack frame on the process's user state stack when

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
			the corresponding exception occurs.
\$370	P\$SigLvl	b	Signal mask nesting level. If this field is non-zero, signals are masked for the process. In this case, if a signal is received the process is still made active (if it was not already), and the signal is put in the signal queue, but the process's signal intercept handler is not called. When the process clears this field (using the F\$SigMask system call), the signal intercept handler is immediately called for each pending signal. This field can be incremented, decremented, or cleared by the F\$SigMask system call.
\$371	P\$SigFlg	b	This field is a set of bit flags for the signal mechanism. Currently, only bit 7 is used. It is set when the process receives a signal while active, and cleared when the process goes to sleep, or returns to user state. If this flag is set, a process can go to sleep even if a signal is pending. This allows a system call to go to sleep, waiting for a signal from (for example) an interrupt service routine, even though another signal is pending.
\$372	P\$Sigxs	w	Number of free entries in the signal queue.
\$374	P\$SigMask	1	This field is a set of 32 bit flags, each bit (0 to 31) corresponding to the signal code of the same number. If the bit is set, then a signal of that code sent to the process is ignored - the process is not woken, and the signal is not queued. Signals 0 (kill) and 1 (wakeup) cannot be filtered in this way. Due to the coding of the F\$Send system call, signal code 32 will be ignored if bit zero of this field is set. There is no system call to alter this field.
\$378	P\$SigCnt	1	Number of signals pending.
\$37C	P\$SigQue	1	Address of the signal queue element containing the next signal to process (the oldest pending signal). The signal queue is a doubly linked list of structures containing one signal code each. The queue is arranged as a ring, so that the "next" pointer of the last entry points to the first entry (oldest pending signal). As each signal is processed (the process's signal handler is called), the kernel clears the signal code field of the entry, and advances this field to point to the next entry in the queue. While queue structures may be cleared (signal code field is set to zero, indicating no signal pending), they are not removed from the queue, and their memory is not freed until the process dies. This field is zero until the process receives a signal.
\$380	P\$DefSig	1 4	Initial signal queue structure. When the process first receives a signal this field is installed as the first and

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$380	P\$DefSig	1 4	Initial signal queue structure. When the process first receives a signal this field is installed as the first and only entry in the signal queue. If the queue is full when a signal is received, a signal queue structure is allocated from system memory. Therefore if the process never receives a signal with one already pending, no dynamically allocated structures are needed.
\$390	P\$Thread	1 2	Addresses of the first and last "thread" structures allocated to this process. This is a doubly linked list of nominally general-purpose structures. At present it is used to record outstanding "alarms" installed by the process using the F\$Alarm system call. If the process has no alarms outstanding these fields will be zero.
\$398	P\$frag	1 2	Addresses of the first and last memory fragment structures. This is a doubly linked list of structures identical to the memory colour node structures used to manage the free pool of memory. This linked list identifies the free memory fragments not yet used from the memory allocated to the process. This is necessary because the kernel will take memory from the free pool only in multiples of the system minimum allocatable block size (at least 256 bytes), but is able to allocate to the process in multiples of the process minimum allocatable block size (currently 16 bytes). When a process makes a request for memory, the kernel first attempts to satisfy the request from the fragments identified by this linked list. Only if this fails does the kernel allocate additional system memory to the process.
\$3A0	P\$MOwn	1	Original owner of the primary module of this process. This protects against a program, written by a user who is not a super user, modifying its own module header (which is naturally within its memory map) to set the "module owner" field to zero, and so gain access to resources reserved for members of the super user group. The kernel uses this field to protect against a process changing its user and group to become a super user (see the description of the F\$SUser system call in the chapter on OS-9 System Calls).

The space at the end of the process descriptor for use as the system state stack (the stack used during system calls made by the process) is slightly more than 1k bytes. This is sufficient if all operating system components are written in assembly language. However, more and more device drivers and file managers are being written in C, which uses much more stack. Some file managers find it necessary to allocate a separate, larger system state stack for each process. The next two fields are available for management of such a stack. The current kernel (OS-9 version 2.4.3) does not use these fields:

OS-9 INTERNAL STRUCTURE

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$3AC		b \$454	System state stack. When the process is forked, its system state stack pointer is set to the top of this memory (the end of the process descriptor). This is therefore (automatically) the stack of the process when in system state (that is, when making a system call).

The high byte of the word **P\$State** contains the bit flags that the kernel uses – in conjunction with the byte **P\$QueueID** – to identify the current state of the process. The bit definitions for **P\$State** (high byte) are:

<u>Bit</u>	<u>Description when set</u>
0	The process is dead. All its resources have been de-allocated except for the process descriptor itself, which will be de-allocated once the parent has received the process's exit status by executing a "wait for child" F\$Wait system call, or the parent dies.
1	The process is condemned. When it next would start execution in user state it will be terminated.
4	The memory map permitted to the process has been altered. The SSM must build a new MMU memory map for the process before the process restarts execution in user state.
5	The time slice of the process has expired. When the process would next start execution in user state, the kernel will perform a reschedule of the active processes.
6	The process is in a timed sleep.
7	The process is executing a system call.

The possible values for **P\$QueueID** (as ASCII characters) are:

<u>Character</u>	<u>Description</u>
	(space) no queue state – should not occur.
-	The process is not in a queue, and is not the current process (usually dead).
a	The process is in the active queue, waiting for processor time.
d	The process has been created for debugging, but is not yet in any queue, or has stopped at a breakpoint, or is executing without tracing ("at full speed").
e	The process is in an event queue, waiting on an event.
m	The process is waiting for a buffer – implemented by the F\$MBuf system call of the Internet Support Package (ISP).
s	The process is in the sleep queue, in timed or untimed sleep.
w	The process is in the waiting queue, waiting for a child process to die.

<u>Character</u>	<u>Description</u>
*	The process is the current process. It is not in any queue.
&	The process is the system state debugger, in suspended state.

The field **P\$frag** contains the root and end pointers to a linked list of structures used to keep track of unallocated memory fragments. Because the process could allocate memory of different colours, the list has the same structure as the system free memory colour node list. Each structure identifies the memory area from which it was taken, its colour and priority, and the first and last memory fragments from this area, as a linked list. Thus each process has a list of "locally free" memory areas that have been allocated to it from the free pool (and so are no longer in the free pool), but have not yet been allocated in response to a program request. The colour node structure is described in the section on memory allocation, in the chapter on OS-9 Modules, Memory, and Processes.

The field **P\$Thread** contains the root and end pointers for the linked list of thread structures (or thread "blocks") allocated for this process. The end pointer points to the last structure in the linked list. Currently thread structures are only used for alarms (see the chapter on Inter-process Communication). Each structure represents a pending or cyclic alarm. Remaining thread structures are automatically de-allocated by the kernel when a process dies. The thread structure is shown below.

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$000	T_ID	w	Thread block type identifier. Because threads may be used for other purposes in the future, the alarm functions write a distinctive "magic number" here (45245) when allocating the thread block, and check it when deleting the alarm.
\$002	T_Proc	w	Process ID of the process that created the block.
\$004	T_MSiz	l	Size of memory allocated for the thread block - used when de-allocating the block.
\$008	T_User	l	Group and user number of the process that created the block (the block owner). The System Process temporarily takes on this group and user number when executing the thread function.
\$00C	T_Next	l	With T_Prev , the "next" and "previous" pointers linking this thread block into the linked list rooted in the System Globals (either D_Thread or D_Alarm). Used by the System Process when checking for alarms that have reached their execution time. The linked list is maintained in execution time order.

OS-9 INTERNAL STRUCTURE

\$010	T_Prev	1	See T_Next , above.
\$014	T_Link	1 2	"Next" and "previous" pointers respectively linking this thread block into the linked list rooted in the process descriptor of the creating process (see the field P\$Thread).
\$01C	T_Sys	1 4	Usage depends on thread block purpose. See below for the usage of this field with alarms.
\$02C	T_Regs	1 18	Register stack frame, provided by the calling process for use when executing the thread function. For user state alarms this stack frame is automatically generated by the kernel, and provides the parameters and routine address for calling the F\$Send system call.

The **T_Sys** field is structured as follows for alarm threads:

<u>Offset</u>	<u>Name</u>	<u>Size</u>	<u>Description</u>
\$01C	T_Cycle	1	For cyclic alarms: number of ticks between alarms. Always zero for non-cyclic alarms.
\$020	T_WkTime	1	For cyclic or relative alarms: the time for thread execution, as ticks since system startup (compared with D_Ticks). For absolute time alarms: time of day for thread execution, as seconds since midnight.
\$024	T_WkDate	1	Only used for absolute time alarms - the Julian date for thread execution.

CHAPTER 15

MICROWARE C AND ASSEMBLY LANGUAGE



Programming in C is usually far more productive than programming in assembly language, and produces code that is more readable, portable, and maintainable. However, assembly language is still used where speed and memory efficiency are important. For these reasons, most of OS-9 itself is written in assembly language, (whereas for portability OS-9000 is written in C). C programs wishing to make calls to the OS-9 operating system must use interface functions written in assembly language.

This section describes the interface between C and assembly language under OS-9, and shows how operating system components such as file managers and device drivers can be written in C.

15.1 MICROWARE C

The latest version of the Microware C compiler (version 3.2, supplied with OS-9 version 2.4) is an up-to-date and complete implementation of the C language, including bit fields, enumerated constants, structure assignment, and structure return. There are no limits on the length of symbol names, and the same element name may be used in different structure definitions. However, the additional features from the ANSI C standard are not incorporated. Microware has announced that an ANSI standard C compiler is in development – at the time of writing its release is imminent.

Microware C conforms to the standard laid down by Kernighan and Ritchie in their definitive book "The C Programming Language". A complete set of library functions are provided, giving both UNIX compatibility (as far as possible), and access to OS-9 specific functions. Both "buffered file" functions

(such as **printf()** and **fread()**), and low level path I/O functions (such as **open()** and **read()**) are provided.

15.2 CIO AND MATH TRAP HANDLERS

The most commonly used I/O library functions are also provided in the **cio** trap handler. This saves on memory, disk space, and time loading programs from disk by making the program much smaller – the program makes trap calls to the I/O functions in **cio**, rather than having the code for the functions in the program module itself.

The decision whether to use **cio**, or to use library functions included in the program, is taken at compile (link) time. The '-i' option to the **cc** executive requests that library functions (from 'LIB/cio.l') be used that simply make trap calls to the **cio** trap handler. If this option is omitted, the program is linked with library functions that perform the I/O functions themselves (from 'LIB/clibn.l' or 'LIB/clib.l'). The functions will execute slightly faster than calling the trap handler, but the program will be significantly larger.

Similarly, math functions can be included in the program, or can make trap calls to the **math** trap handler module. If the '-x' option is specified to **cc**, the trap handler technique is used, by linking to the 'LIB/clib.l' library and generating "in-line" trap instructions in the program. Otherwise the 'LIB/math.l' library is used. In addition, if the target processor is indicated as a 68020/030/040 (using the '-k' option of **cc**), the compiler will use the additional integer arithmetic instructions of these members of the 68000 family, and if the '-k2F' option is used to indicate that the 68881/68882 floating point coprocessor is available (or the processor is a 68040), the compiler will use in-line floating point instructions.

15.3 THE REMOTE DIRECTIVE

All OS-9 programs use register indirect indexing (relative to the **a6** register) to access static storage memory. However, the 68000 and 68010 are limited to signed 16-bit constant offset indexing with address registers. The linker automatically offsets data space accesses by 32k bytes, allowing 64k bytes of storage to be accessed using constant offset indexing. To access more static storage than this the compiler must use a different form of addressing. The OS-9 C compiler offers three options to do this.

The first approach uses the **remote** directive, which is an additional keyword in the Microware C language, prefixing static storage definitions and declarations. For example:

```
remote static int x;
```

This declares that **x** is likely to lie outside of the 64k limit, and therefore an alternative addressing mode must be used. The C compiler generates two instructions for each access to such variables, as it must load a data register with the long word offset, and then use register indirect addressing with index. The linker places all **remote** variables after all ordinary static storage, so it is usually sufficient to define only large arrays as **remote**. The **remote** keyword is specific to the OS-9 C compiler, and so its use is not portable.

The second approach is to use the option '-k0L' with the **cc** executive. This instructs the compiler to generate two instructions for each static storage access, as if all static variables had been defined as **remote**. This makes programs more portable (and simplifies the porting of existing programs to OS-9), but also makes the programs slower and larger than necessary.

The third approach is only available with the 68020/030/040 version of the C compiler. The '-k2L' option allows the user to specify that long word constant offset indexing be used for data space accesses - this is an additional addressing mode in the 68020/030/040. This causes all static storage references to use long word offsets, allowing the full use of the 32-bit addressing capability of the 68000 family. However, it does mean that accesses to the first 64k bytes of the program's static storage are using 32-bit offsets when 16-bit offsets would do, making the program longer and slower.

As most programs do not need more than 64k bytes of static storage (large buffers are instead dynamically allocated, and addressed via pointers), these techniques are generally not required.

The same problem applies to function calls in programs larger than 32k bytes, as the 68000 **bsr** instruction is limited to a signed 16-bit offset. The linker therefore uses a jump table in the data space for all references that exceed +/-32k bytes; it patches over the **bsr** instruction generated by the compiler with a **jsr** relative to the data space. The 68020/030/040 version of the C compiler allows the user instead to specify (using the '-k2CL' option) that long word offsets be used with the **bsr** instruction, as this is available on the 68020/030/040 processors. This may be advantageous for very large programs, but bear in mind that all function calls will then use long word offsets, creating a larger and perhaps slower program.

Similarly, the '-k0CL' option causes the C compiler to generate two instructions for every function call instead of a simple **bsr** instruction. This approach is compatible with the 68000 and 68010, which do not support long word **bsr** offsets.

Note that if any of the **cc** '-k2' options are used, causing the compiler to generate code using the additional instructions and addressing modes available on the 68020/030/040, the program cannot be run on a 68000 or 68010.

15.4 PROGRAM STARTUP

The C programmer expects execution to start with his **main()** function. However, because the kernel passes parameters to a newly forked program in the processor registers, it is necessary for an assembly language "core" to be called first, which then calls **main()**. This "core" is in the file 'C/SOURCE/cstart.a'. It is provided already assembled in the file 'LIB/cstart.r'. The **cc** executive automatically makes this the first Relocatable Object File (ROF) in the linker command line it generates. 'cstart.r' also is the only file in a C program that has a "non-null" ("root") **psect** directive. The **psect** assembler directive specifies the output module type, attributes, and revision number. The linker permits only one root **psect**, so files produced by the C compiler have null **psect** directives.

If **cc** is used to generate the linker command line, 'cstart.r' is automatically included. However, if you create your own linker command line you must manually include 'cstart.r' as the first ROF in the command line. Try using the '-bp' option of the **cc** executive to display the command lines that **cc** generates.

15.5 C WITH ASSEMBLY LANGUAGE

The problems in interfacing C to assembly language are to do with parameter passing and register usage. Provided that the C compiler's parameter passing and register usage schemes are understood, assembly language functions can be written that are called from C functions, or that call C functions.

The C compiler uses a simple and consistent parameter passing mechanism. The first two long words of parameters are passed to a function in the **d0** and **d1** registers. If the first parameter is a double precision floating point number it is passed in **d0** and **d1**. If the first parameter is not a double, but the second is, **d1** is not used. The remaining parameters are on the stack

(above the return address). If a function returns a value, the returned value is in **d0** (and **d1** if the returned value is a **double**). If the returned value is a structure, its *address* is returned in **d0** – the structure itself is built in static storage private to the called function (as described below).

The only register usage convention that the assembly language programmer needs to know (and can rely on) is that the address of the program static storage is always held in the **a6** register.

Because the compiler's use of registers cannot be relied on (future releases of the compiler may behave differently), you should never embed assembly language within C functions. Instead, always use separate assembly language subroutines, called as functions from the C code. For the same reason, the assembly language subroutine should always preserve all of the processor registers other than **ccr**, **d0**, and **d1**. Indeed, even **d0** and **d1** should be preserved if they are not used to pass a parameter or return a value.

To increase the portability of your C code, the assembly language functions should be placed in a separate source file. Only this file will need modification if your C source code is ported to a different processor or compiler. Note that if an assembly language symbol is to be used from a different source file, it must be made public. The Microware assembler makes a symbol public if its name is immediately followed by a colon in the line defining the symbol.

15.6 REGISTER VARIABLES

The C language provides for automatic variables (temporary variables within functions) and parameters passed to functions to be declared as **register**. The C compiler will try to place these variables in processor registers, rather than in memory allocated from the stack. Because such variables do not have to be read from memory when their value is required, or written to memory when their value is changed, code fragments using such variables are much smaller and faster. This is particularly important if the variable is used frequently – for example, a pointer to **char** used for parsing a string, or an **int** used as the controlling variable in a **for** loop.

The C language specification states that the compiler will assign register variables to processor registers in the order in which the definitions appear. Once all the available processor registers have been used, further **register** definitions will create ordinary automatic variables on the stack. Therefore the order in which the register variables are defined is very important, especially if the code is intended to be portable – another processor may have

fewer registers, and another compiler may make fewer registers available for register variables.

The Microware C compiler makes four data registers (**d4-d7**) and three address registers (**a2-a4**) available for register variables. The data registers are used for register variables of types **char**, **short**, **int**, and **long** (and the unsigned equivalents). The address registers are used for pointers of any type. If the '-k2F' option of the **cc** executive has been used to indicate that a floating point unit is available, the C compiler makes six FPU registers (**fp2-fp7**) available for register variables of type **double** or **float**.

15.7 CODING FOR SPEED

It is a widely stated axiom that a program spends most of its time executing only a very small percentage of its code in a frequently repeating loop. By carefully identifying such code fragments and making judicious use of register variables within them, the programmer can make his program run significantly faster.

The following example of a function to copy a string executes very much faster because register variables have been used. Try compiling this function with the '-a' option of the **cc** executive, and study the output with and without the use of the **register** keyword. Using the '-c' option as well will keep the C source code as comments in the assembly language file. Notice that the Microware C compiler makes use of special features of the 68000 instruction set, such as the post-increment addressing mode, to reduce the code length and increase the speed.

```
void strcpy(d,s)
register char *d,          /* string to copy to */
            *s;           /* string to copy from */
{
    do {
    } while ((*d++=*s++)!='\0');
}
```

Because the C compiler must not only conform to the C language specification, but must also make use of the 68000 instruction set as provided by Motorola, careful use of certain coding techniques can also significantly improve execution speed. For example, the two fragments shown below produce the same effect (copy a given number of bytes):

```
void copybytes(d,s,n)
register char *d,          /* where to copy to */
            *s;           /* where to copy from */
register int n;           /* number of bytes to copy */
```

```

{
#ifdef FIRST_METHOD
    while (n--)
        *d++=*s++;
#else
    if (n!=0) {
        do {
            *d++=*s++;
        } while (--n);
    }
#endif
}

```

However, the second method will execute much more rapidly. In the first method, to comply with the C language specification, the register variable **n** must be saved (the Microware compiler saves it to **d0**) and then decremented, and then the saved value must be tested, and a conditional branch taken on the result – four instructions, plus one for the instruction to copy the byte, making five in total for the loop. In the second method, a byte is copied, then the register variable is decremented, and a conditional branch taken on the result – a total of three instructions. Thus the second method will execute significantly faster. The extra instruction `if (n!=0)` is unimportant for the speed of the function, because it is executed only once.

If program execution speed is important, you should identify the small percentage of the program that is consuming most of the processor's time. If you are already familiar with the behaviour of the C compiler, you can then recode those fragments for optimum speed of execution. Otherwise, use the '-a' option of the **cc** executive to compile the C source code to assembly language, and study the behaviour of the C compiler in those critical code fragments. This should help you decide on appropriate modifications to the C source code.

If, after having optimized your C source code in this way, you feel that the output of the C compiler is still not producing the fastest code possible, you can write a separate assembly language function to be called instead of the C code. Of course, this should be a last resort, as it is less portable and less readable.

A corollary to the axiom that the program spends most of its time executing only a small part of the code, is that the error handling parts of the program are executed very infrequently – most of the time a program does not generate errors. Therefore it is reasonable to allow error handling code to be designed to be easy to write, and informative in its output, rather than trying to code it carefully for rapid execution. This applies to other parts of the

program that will be used infrequently (perhaps only once), such as command line parameter parsing.

15.8 THE 'LINK' INSTRUCTION

The C compiler uses the 68000 **link** instruction to capture the stack pointer on entry to a function. This is required by the Microware C Source Level Debugger, to give access to the stack frame (parameters passed, and automatic variables) during execution of the function (for example at a breakpoint or when single stepping).

Note that because the Source Level Debugger has no information about the parameter structure or stack allocation for variables in a function written in assembly language, the **link** and **unlk** instructions are not strictly needed when writing a function in assembly language, although including them allows the **frame** command of **srcdbg** (and the **w** command of **debug**) to report which function called the assembly language function.

The following example shows typical code produced by the C compiler. The C function:

```
set5(p)
int *p;
{
    int x;
    x=5;
    *p=x;
}
```

produces the following assembly language when compiled (I have added the comments!):

set5:	link	a5,#0	stack a5, put sp in a5, add 0 to sp
	movem.l	d0/a0,-(sp)	save parameter and a0 register
	move.l	#-68,d0	ensure at least 68 bytes stack free
	bsr	_stkcheck	
	subq.l	#4,sp	allocate automatic: x
	moveq.l	#5,d0	set x = 5
	move.l	d0,(sp)	
	movea.l	4(sp),a0	get parameter: p
	move.l	(sp),(a0)	copy x to *p
	addq.l	#4,sp	de-allocate x
	movem.l	-4(a5),a0	retrieve a0 register
	unlk	a5	put a5 in sp, unstack a5
	rts		

Note the use of the stack for automatic variables, and for temporary storage. The **_stkcheck()** function is called to determine that enough stack space is

available for the function's needs. If not, the `_stkcheck()` function exits the program with a `*** stack overflow ***` message. The `_stkcheck()` function is part of 'cstart.a', and uses static storage variables initialized by 'cstart.a' to determine whether there is sufficient stack available. Because 'cstart.a' is only used when creating a program, the `_stkcheck()` function is not appropriate for the creation of operating system components such as device drivers and file managers – either stack checking must be disabled using the '-s' option of `cc`, or the programmer must supply an alternative `_stkcheck()` function.

15.9 A FUNCTION IN ASSEMBLY LANGUAGE

One common reason for writing a C-callable function in assembly language is to supplement the standard C libraries. Microware have not supplied library functions for the privileged (system state only) system calls. The example below shows a typical assembly language function to make the **F\$IRQ** system call, with an example of this function being called from C:

```
{
    if ((f_irq(vector,priority,handler,port))==ERROR)
        printf("Error #%d\n",errno);
}
```

This call to the function `f_irq` is passing four parameters. The first parameter (the interrupt vector number) will be in the `d0` register, and the second parameter (the software polling priority) will be in the `d1` register. The remaining two parameters (the address of the interrupt handler function, and the address of the interface) will be on the stack. The third parameter will be just above the return address (at `4(a7)`), while the fourth parameter is above that (at `8(a7)`).

The following implementation of `f_irq` makes good use of the fact that the parameters have been ordered so that `d0` and `d1` already contain the correct parameters for the **F\$IRQ** system call, and the remaining two parameters are in the correct order to be picked up by a single `movem` instruction. As always, the `a6` register contains the static storage address, which must be copied to the `a2` register as a parameter to the **F\$IRQ** system call.

The function returns 0 if the **F\$IRQ** call gave no error. Otherwise, the error code is extended to a long word and saved in the static storage field `errno`, and -1 is returned. Note that the returned value is in the `d0` register, as required by the C compiler. The function saves all the registers that it modifies (including `d1`, which will be modified if **F\$IRQ** returns an error). This takes 16 bytes of space on the stack, so the third parameter passed to

the function is at **20(a7)** – 16 bytes of temporary storage, plus 4 bytes of return address.

```
#asm
f_irq:      movem.l d1/a0/a2-a3, -(a7)   save registers
* d0 and d1 already contain the correct parameters
            movem.l 20(a7), a0/a3       get handler and port addresses
            movea.l a6, a2              copy static storage address
            os9      F$IRQ
            bcs.s   f_irq10             ..error
            moveq   #0, d0              no error - return 0
            bra.s   f_irq20
f_irq10     move.l  d1, errno(a6)        save OS-9 error code
            moveq   #-1, d0             indicate error
f_irq20     movem.l (a7)+, d1/a0/a2-a3   retrieve registers
            rts
#endasm
```

Note that the symbol **f_irq** is terminated by a colon. This causes the assembler to make the symbol public, so the function can be called from another source file. Of course, it is also good practice to provide a proper declaration for the function in your C source code. Note also that although the OS-9 error codes are 16-bit values, the kernel ensures error codes returned by system calls are 32-bit, with the high word set to zero.

15.10 STRUCTURE RETURN

A structure can be of any length. Therefore it is not possible to use processor registers to return a structure. Instead, if a function is defined as returning a structure, the compiler reserves a private block of static storage for the returned structure, and returns a pointer to the structure. The following example illustrates parameter passing with structure return. It shows a C program, and the output of the C compiler in assembly language. This example also shows the use of register variables by the C compiler.

```
#include <stdio.h>
typedef struct {                /* declare a structure type */
    int x,y,z;
} int3;

int3 setints(); /* declare a function returning an item of that type */

main()
{
    int3 n;                    /* define an automatic of that type */
    n=setints(4,5,6);          /* set it from the value returned */
    printf("%d %d %d\n", n.x, n.y, n.z); /* display the results */
}
```



```

/* Define the function returning an item of the structure type: */
int3 setints(i,j,k)
register int i,j,k;
{
    int3 m;           /* define an automatic of that type */
    m.x=i;            /* fill it with the given values */
    m.y=j;
    m.z=k;
    return(m);        /* return the structure */
}

```

The following assembly language output was obtained using the command:

```
$ cc -qixa test1.c
```

The '-a' option instructs the **cc** executive to save the assembly language output of the C compiler in a file 'test1.a', and not to proceed to assemble and link the program. Of course, the C compiler does not generate comments for its assembly language output – the comments are my addition!

```

                psect    test1_c,0,0,0,0,0    "null" psect starts code file
                nam      test1_c              nam and t1l are listing directives
                t1l      main
* Note that symbols not declared with 'static' are public:
main:          link     a5,#0                 save a5, set stack frame ptr
                movem.l  #_111,-(sp)           save registers d0-d1/a0
                move.l   #_3,d0               check 88 bytes of stack free
                bsr      _stkcheck
                lea      -12(sp),sp           allocate stack for 'n'
                pea      6,w                 third parameter is 6
                moveq.l  #5,d1               second parameter is 5
                moveq.l  #4,d0               first parameter is 4
                bsr      setints             call function 'setints'
                addq.l   #4,sp               ditch third parameter
                movea.l  d0,a0               copy returned value - ptr to structure
                move.l   (a0),(sp)           copy the structure to 'n'
                move.l   4(a0),4(sp)         three long words
                move.l   8(a0),8(sp)
                move.l   8(sp),-(sp)         fourth parameter is 'n.z'
                move.l   8(sp),-(sp)         third parameter is 'n.y'
                move.l   8(sp),d1           second parameter is 'n.x'
                lea      _5(pc),a0          point at format string
                move.l   a0,d0              it is the first parameter
                bsr      printf             call function 'printf'
                addq.l   #8,sp              ditch parameters on stack
                lea      12(sp),sp           de-allocate stack used for 'n'
_4             movem.l  -8(a5),#_1          retrieve registers d1/a0
                unlk     a5                 restore a5 and stack ptr
                rts                                     return to 'cstart'
_3             equ      0xffffffffa8
_1             equ      0x00000102
_2             equ      0x00000014

```

```

* Function 'setints'. Note that the 'link' instruction saves a5 on the
* stack, so after 6 other registers have been saved, the third
* parameter is at 32(a7). Notice also how the compiler makes use of
* the d4, d5, and d6 registers for 'register' variables:
setints:    link    a5,#0          save a5, set stack frame ptr
            movem.l #_6!3,-(sp)    save registers d0-d1/a0-a2/a4
            move.l  d0,d4          copy first parameter to register
            move.l  d1,d5          copy second parameter to register
            move.l  0+_7(sp),d6    copy third parameter to register
            move.l  #_8,d0         check 76 bytes of stack free
            bsr     _stkcheck
            vsect
            align
            ds.b12                start local static storage
            return                force word alignment
_10         end                    reserve storage for structure
*          end local static storage
            lea     -12(sp),sp      allocate stack for 'm'
            move.l  d4,(sp)        'm.x' = first parameter
            move.l  d5,4(sp)       'm.y' = second parameter
            move.l  d6,8(sp)       'm.z' = third parameter
            move.l  (sp),_10(a6)    copy 'm' to local static storage
            move.l  4(sp),_10+4(a6) three long words
            move.l  8(sp),_10+8(a6)
            lea     _10(a6),a0      point at the local structure
            move.l  a0,d0           it is the returned value
            lea     12(sp),sp      de-allocate stack used for 'm'
            bra     _9
            nop
_9          movem.l -16(a5),#_6      retrieve registers a0-a2/a4
            unlk    a5             restore a5 and stack ptr
            rts                  return to 'main'
_8          equ     0xffffffffb4
_6          equ     0x00000170
_7          equ     0x00000020
_5          dc.b"%d %d %d", $d,$0  the 'printf' format string
            ends                  end of code

```

15.11 CALLING C FROM ASSEMBLY LANGUAGE

The OS-9 kernel is written in assembly language, and device drivers and file managers have traditionally also been written in assembly language. Therefore the operating system interface to file managers and device drivers uses processor registers for parameter passing, and is not directly compatible with the output of the C compiler. However, file managers, device drivers, and other operating system components can be written in C, provided an appropriate assembly language "skeleton" is used. This can (and, in fact, must) be in a separate source file, so the programmer need only be concerned with writing in C. Once the appropriate skeleton has been written, it can be

used without modification for other device drivers (or file managers, and so on).

As described above, the C compiler stack checking routine is not compatible with the system state stack usage within the operating system, so the '-s' option of the **cc** executive must be used, to instruct the C compiler not to make calls to the **_stkcheck()** function to check for free stack space. Alternatively, the programmer can write his own **stkcheck()** function, to check for stack overflow within the system state stack (which uses the second half of the process descriptor). If this is done, any interrupt service routine must be in a separate source file which is compiled with stack checking disabled, because a different stack (the interrupt stack) is used during interrupt processing.

If variables are defined as static storage (either because they are defined in the outermost scope of the source file, or because they are prefixed with the **static** keyword), the C compiler generates a **vsect** to indicate to the linker that static storage is required, and generates instructions using addressing relative to the **a6** register to access them. When creating the final output module, the linker adds up all the static storage definitions from **vsect** sections, and puts the total in the "memory requirement" field of the module header (**M\$Mem**). The linker also adjusts all instructions that address static storage locations, to take account of the previously allocated static storage from previous source files. Lastly, if the output module is of type "trap handler" or "program", the linker adjusts all static storage addressing references by -32k bytes, as the kernel adds 32k bytes to the static storage pointer register (**a6**) when forking a program or calling a trap handler. This helps compensate for the signed 16-bit constant offset indexing limitation of the 68000 (as described above).

The C compiler is intended for the production of programs, so it generates static storage variable references using the **a6** register. When writing an operating system component, such as a device driver or file manager, C static storage definitions can be used to provide references into a chosen memory structure, provided the address of that memory structure is placed in the **a6** register by the assembly language "skeleton". If the "memory requirement" field (**M\$Mem**) of the module header of the operating system component is not used, any memory structure can be chosen. For example, a file manager might use static storage references to access the path descriptor fields. Otherwise, the chosen memory structure must be the structure whose size is determined by the "memory requirement" field, because the linker generates this field by adding the sizes of all the static storage definitions (**vsects**). For example, this field in the header of a device driver is used by the kernel to

determine the size of the Device Static Storage to be allocated. Therefore static storage definitions and declarations in a device driver must refer to variables in the Device Static Storage, just as they would if the device driver were written in assembly language.

An important consideration when writing operating system components in C is the amount of stack used. During a system call, the second half of the process descriptor is used for the stack. This gives a total of about 1k bytes of stack. During an I/O call this will be used by the kernel, the file manager, and the device driver, and by any system calls that these components make themselves. It is easy to use up a lot of stack when writing in C. The compiler uses stack for preserving registers, for automatic variables, and for passing parameters when calling other functions. Because a stack overflow in system state is catastrophic (the upper part of the caller's process descriptor will be corrupted), it is important to be sparing in the use of C features that will cause stack usage. Stack space can be saved by minimizing the number of levels of nesting of function calls, and by using static storage variables rather than automatic variables and function parameters. Using register variables does not save stack space, as the compiler will save the current contents of the registers onto the stack on entry to the function.

If you suspect that stack overflow may be a problem, you can add a system state stack checking function, as shown in the example skeleton for a file manager below. Also, if you set a breakpoint (using the system state debugger) within the device driver, you can display the process descriptor memory, and see if the stack usage is approaching the top of the process descriptor variables structure. As a last resort, if you need more stack, you can add a stack switching capability to the device driver or file manager skeleton, perhaps using the field **P\$ExpStk** in the Process Descriptor (see the section on the Process Descriptor in the chapter on OS-9 Internal Structure). This uses a stack space allocated by the driver when it is initialized, or by the file manager when the path is opened. The skeleton routine saves the current stack pointer (perhaps in the Device Static Storage or **P\$ExpStk**), and then sets the stack pointer to the top of the allocated stack space. On return from the C function, it restores the original stack pointer.

If this technique is used, it is important to be sure that there cannot be concurrent calls that would use the same stack. For example, there cannot be concurrent calls on the same path (the kernel queues such calls), so a stack space allocated when a path is opened is secure against concurrent usage. Similarly, SCF and RBF queue concurrent calls into the device driver, by marking the Device Static Storage (except for SCF Get Status calls).

Therefore a device driver could use a stack space allocated during its initialization routine. However, a file manager should not use such a stack space, as there may be multiple concurrent calls to the file manager for the same device, but on separate paths. A file manager must therefore use a stack allocated for each path.

The problem of concurrent access can be completely relieved by allocating a stack extension to each process, saving the address and size in the **P\$ExpStk** field of the process descriptor. If a stack extension is allocated for each process, the stack allocated must be sufficient for all file managers and device drivers needing extra stack space that the process may call. Note that the kernel does not use the **P\$ExpStk** field. If a stack extension is allocated using this field, an extension to the **F\$DelTsk** system call must be added (perhaps in a kernel customization module) to check this field and de-allocate any stack extension when the process is terminated. Similarly, an extension to the **F\$AltTsk** system call could be added to allocate the stack extension when the process is created, rather than leaving it to the discretion of individual file managers and device drivers. Note that if this technique of "global" stack switching is used, any operating system components that perform stack checking by expecting the system state stack to be in the process descriptor will fail.

15.12 A DEVICE DRIVER IN C

As described above, a device driver written in C requires an assembly language "skeleton" file. This skeleton has four main functions which cannot be provided from a C source file:

- a) Provide a root **psect**, giving the module type and the address of the table of routine offsets to the linker.
- b) Generate the table of offsets to the device driver routines (the routines in the skeleton).
- c) In each routine, convert the parameters passed by the kernel or file manager into a form suitable for passing to a C function, and then call the C function.
- d) In each routine, on return from the C function, convert the returned error status into the standard OS-9 format expected by the kernel or file manager, and convert any returned values into the format expected by the kernel or file manager.

The device driver skeleton shown below is for a device driver working with the RBF file manager. It can be used for any RBF device driver. A different skeleton is required for each file manager, because the parameters passed are different, but the skeleton below can easily be adapted to any file manager – the principles are the same. In the skeleton below, the `a5` register is reset to zero on entry to each function, to indicate to a source level debugger that this is the top of the available stack frame. The instructions are not strictly necessary unless a source level debugger is going to be used to debug the driver. The present Microware Source Level Debugger cannot be used to debug operating system components, although at the time of writing a system state source level debugger is under development.

In addition to the skeleton, the device driver writer will also need to write assembly language functions to permit the C code to make any necessary operating system calls. This applies to the privileged (system state only) calls, for which there are no functions in the Microware C library, but also to the non-privileged calls. This is because many of the standard C library functions assume that they are being called from a user state program, and use variables and buffers that are not appropriate to a device driver or other operating system component. However, the math functions (in the library `'dd/LIB/math.l'`) can be used, and indeed calls to these functions will automatically be generated by the C compiler.

The skeleton shown below takes no account of the use of floating point registers in an FPU. This is not necessary because the C compiler generates instructions at the start of each function to save any FPU registers it will use, and at the end of the function to restore them.

The C language specifies that static storage variables can be defined with initializing values. When a program is forked under OS-9, the kernel uses information in the module header to initialize any such variables in the program's static storage. However, the kernel does not perform such a function when allocating a Device Static Storage (or any other operating system memory structure), so initialized static storage variables cannot be used. Note that the kernel *does* clear the Device Static Storage to zeros before calling the initialization function of the device driver.

Because the kernel does not initialize any static storage variables, the "jump table" created by the linker for accessing subroutines at a relative distance exceeding $\pm 32k$ bytes cannot be used. To ensure that the linker has not found the need to generate a jump table, the `'-a'` option of the linker should not be used. If the device driver is larger than 32k bytes in size, the `'-k2cl'` option of the `cc` executive can be used, provided the target processor is a

68020/030/040, or the '-k0cl' option if the target processor is a 68000/010/070. Otherwise some manually-coded technique must be used to overcome this limitation of the 68000 processor.

□ RBF Device Driver Skeleton

```

* File 'rbskel.a'
* Device driver skeleton for RBF device drivers
    use      /dd/DEFS/oskdefs.d
Typ_Lang    set      (Drivr<<8)+0b_jct
Attr_Rev    set      ((ReEnt+SupStat)<<8)+0
Edition     set      1
* The psect directive, giving the module type as "device driver":
    psect    rbskel,Typ_Lang,Attr_Rev,Edition,0,EntryTable
* The table of offsets to the driver routines:
EntryTable  dc.w      Init          initialize
            dc.w      Read          read
            dc.w      Write        write
            dc.w      GetStat      get status
            dc.w      SetStat      set status
            dc.w      Term         terminate
            dc.w      0            (exception handler)
*****
* Init
*   Initialize device driver
*
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int init(dd)
*           mod_dev *dd;           device descriptor ptr
*
Init:
    movea.w #0,a5                reset stack frame ptr
    move.l  a6,sysglobs(a2)      save the System Globals ptr
    move.l  a4,procdesc(a2)      save the Process Descriptor ptr
    move.l  a2,a6                copy the static storage ptr
    move.l  a1,d0                pass the device descriptor ptr as
*                               the first (only) parameter
    bsr     init                 call the C function
    move.l  d0,d1                copy returned error code
*                               (0 => no error)
    beq.s   Init90               ..no error; carry is clear
    ori     #Carry.ccr           set carry to indicate error
Init90    rts

```

MICROWARE C AND ASSEMBLY LANGUAGE

```

*****
* Read
*   Read sectors
*
* Passed:  d0.l = number of sectors to read
*          d2.l = start LSN
*          (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor
*          (a6) = System Globals
*
* Returns: carry set if error, with error code in d1.w
*
* Calls C function:
*       int read(n,b)
*       unsigned int n,          number of sectors to read
*       b;                      first sector to read
*
Read:
    movea.w #0,a5          reset stack frame ptr
    move.l  a1,pathdesc(a2) save the Path Descriptor ptr
    move.l  a4,procdesc(a2) save the Process Descriptor ptr
    move.l  a2,a6           copy the static storage ptr
    move.l  d2,d1           pass start LSN as second parameter
* The following line is needed prior to OS-9 version 2.4:
    andi.l  #$000000ff,d0   make sector count a long word
    bsr     read            call the C function
    move.l  d0,d1           copy returned error code
*                          (0 => no error)
    beq.s   Read90          ..no error; carry is clear
    ori     #Carry,ccr      set carry to indicate error
Read90
    rts
*****
* Write
*   Write sectors
*
* Passed:  d0.l = number of sectors to write
*          d2.l = start LSN
*          (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor
*          (a6) = System Globals
*
* Returns: carry set if error, with error code in d1.w
*
* Calls C function:
*       int write(n,b)
*       unsigned int n,          number of sectors to write
*       b;                      first sector to write
*

```


Write:

```

movea.w #0,a5          reset stack frame ptr
move.l  a1,pathdesc(a2) save the Path Descriptor ptr
move.l  a4,procdesc(a2) save the Process Descriptor ptr
move.l  a2,a6           copy the static storage ptr
move.l  d2,d1           pass start LSN as second parameter
bsr     write           call the C function
move.l  d0,d1           copy returned error code
*                               (0 => no error)
      beq.s  Write90     ..no error; carry is clear
      ori    #Carry,ccr  set carry to indicate error

```

Write90

rts

* GetStat

* Get Status wild card call

*

* Passed: d0.w = function code

* (a1) = Path Descriptor

* (a2) = Device Static Storage

* (a4) = Process Descriptor

* (a6) = System Globals

*

* Returns: carry set if error, with error code in d1.w

*

* Calls C function:

* int getstat(c,r)

* unsigned int c;

function code

* REGISTERS *r;

ptr to caller's stack frame

*

GetStat:

```

*      move.l  a5,d1          pass caller's register stack
                                frame ptr as second parameter
movea.w #0,a5          reset stack frame ptr
move.l  a1,pathdesc(a2) save the Path Descriptor ptr
move.l  a4,procdesc(a2) save the Process Descriptor ptr
move.l  a2,a6           copy the static storage ptr
andi.l  #$0000ffff,d0    make function code long
bsr     getstat         call the C function
move.l  d0,d1           copy returned error code
                                (0 => no error)
      beq.s  GetStat90     ..no error; carry is clear
      ori    #Carry,ccr  set carry to indicate error

```

GetStat90

rts

```

*****
* SetStat
*   Set Status wild card call
*
* Passed:   d0.w = function code
*           (a1) = Path Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int setstat(c,r)
*           unsigned int c;           function code
*           REGISTERS *r;             ptr to caller's stack frame
**
SetStat:
    move.l   a5,d1           pass caller's register stack
                             frame ptr as second parameter
*
    movea.w  #0,a5           reset stack frame ptr
    move.l   a1,pathdesc(a2) save the Path Descriptor ptr
    move.l   a4,procdesc(a2) save the Process Descriptor ptr
    move.l   a2,a6           copy the static storage ptr
    andi.l   #$0000ffff,d0   make function code long
    bsr      setstat         call the C function
    move.l   d0,d1           copy returned error code
*
                             (0 => no error)
    beq.s    SetStat90       ..no error; carry is clear
    ori      #Carry,ccr      set carry to indicate error

SetStat90
    rts
*****
* Term
*   Terminate device driver
*
* Passed:   (a1) = Device Descriptor
*           (a2) = Device Static Storage
*           (a4) = Process Descriptor
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Calls C function:
*           int term(dd)
*           mod_dev *dd;         device descriptor ptr
*
Term:
    move.l   a6,-(a7)         save register
    movea.w  #0,a5           reset stack frame ptr
    move.l   a4,procdesc(a2) save the Process Descriptor ptr

```

```

        move.l  a2,a6          copy the static storage ptr
        move.l  a1,d0          pass the device descriptor ptr
*                               as the first (only) parameter
        bsr     term           call the C function
        move.l  d0,d1          copy returned error code
*                               (0 => no error)
        beq.s   Term90         ..no error; carry is clear
        ori     #Carry,ccr     set carry to indicate error
Term90
        movea.l (a7)+,a6       retrieve register
        rts
*****
* IRQSvc
*   Interrupt service routine. It is assumed that the C 'init()'
*   function has installed this routine as the interrupt handler, using
*   the F$IRQ system call.
*
*   Passed:   (a2) = Device Static Storage
*             (a3) = Port Address
*             (a6) = System Globals
*
*   Returns:  carry set if the interrupt is not from our device
*
*   Calls C function:
*       int irqsvc(port)
*       void *port;           port address
*
IRQSvc:
        move.l  a5,-(a7)       save register
        movea.w #0,a5          reset stack frame ptr
        move.l  a2,a6          copy the static storage ptr
        move.l  a3,d0          pass the port address as the
*                               first (only) parameter
        bsr     irqsvc         call the C function
        tst.l   d0             was interrupt handled?
        beq.s   IRQSvc90       ..yes; carry is clear
        ori     #Carry,ccr     set carry to indicate not our
*                               interrupt
IRQSvc90
        movea.l (a7)+,a5       retrieve register
        rts
        ends

```

One or more separate C source files must be created, containing the code to perform the actual work of the device driver. The example below shows a "null" driver, compatible with the RBF driver skeleton shown above. Each function returns zero if there was no error, otherwise it returns the appropriate OS-9 error code. Note that the structure **rbfs** (**struct rbfstatic**) defines the kernel and file manager static storage, including the drive tables. This structure type is declared in 'DEFS/rbf.h'. Because this static storage is

defined within the source code of the device driver, the static storage files such as 'LIB/drvs4.l' are not needed (and must not be used) at link time.

```

/* RBF device driver in C. Must be linked with 'rbskel.r'.
   Operating system definitions: */
#define RBF_MAXDRIVE 4          /* number of drives (required by
                                'rbf.h') */
#include <errno.h>              /* error codes */
#include <modes.h>              /* file access modes */
#include <rbf.h>                /* RBF structures */
#include <MACHINE/reg.h>        /* register stack frame */
#include <procid.h>             /* process descriptor */
#include <sg_codes.h>           /* Get/Set status codes */
#include <path.h>               /* common path descriptor structure */
/* Functions in 'rbskel.a':
extern int IRQSvc();           /* interrupt handler skeleton */
/* Static storage definitions (for Device Static Storage): */
struct rbfstatic rbfs;        /* kernel and file manager static */
void *sysglobs;               /* System Globals ptr */
Pathdesc pathdesc;           /* path descriptor ptr */
procid *procdesc;             /* process descriptor ptr */
int errno;                    /* general error number storage */
unsigned short irqmask;       /* status register image for masking
                                interrupts */

/* Initialize */
int init(dd)
mod_dev *dd;                  /* pointer to device descriptor */
{
    rbfs.v_ndrv=RBF_MAXDRIVE; /* set number of drives supported */
    irqmask=dd->_mirqlvl<<8 | 0x2000; /* build status register image
                                        for masking interrupts */
    /* Install interrupt handler: */
    if (f_irq(dd->_mvector,dd->_mpriority,IRQSvc,rbfs.v_port)==ERROR)
        return(errno);        /* error - return error code */
    return(0);                 /* no error */
}
/* Read sectors */
int read(n,s)
unsigned int n,                /* number of sectors to read */
s;                             /* start LSN */
{
    register Rbfdrive dtb;      /* drive table ptr */
    register struct rbf_opt opts; /* path descriptor options ptr */
    unsigned char *buffer;      /* ptr to buffer to read into */
    int drvnum;                 /* logical drive number */
    dtb=pathdesc->rbfpvt.pd_dtb; /* get drive table ptr */
    buffer=pathdesc->path.pd_buf; /* get buffer ptr */
    opts=&pathdesc->rbfopt;      /* point at path descriptor options */
    drvnum=opts->pd_drv;         /* get logical drive number */
    return(0);                  /* no error */
}

```

```

/* Write sectors */
int write(n,s)
unsigned int n,          /* number of sectors to write */
s;                      /* start LSN */
{
    register Rbdrive dtb; /* drive table ptr */
    register struct rbf_opt opts; /* path descriptor options ptr */
    unsigned char *buffer; /* ptr to buffer to write from */
    int drvnum; /* logical drive number */
    dtb=pathdesc->rbfpvt.pd_dtb; /* get drive table ptr */
    buffer=pathdesc->path.pd_buf; /* get buffer ptr */
    opts=&pathdesc->rbfopt; /* point at path descriptor options */
    drvnum=opts->pd_drv; /* get logical drive number */
    return(0); /* no error */
}

/* Get status */
int getstat(f,r)
int f, /* function code */
REGISTERS *r; /* ptr to caller's register stack frame */
{
    register Rbdrive dtb; /* drive table ptr */
    register struct rbf_opt opts; /* path descriptor options ptr */
    dtb=pathdesc->rbfpvt.pd_dtb; /* get drive table ptr */
    opts=&pathdesc->rbfopt; /* point at path descriptor options */
    switch (f) { /* act according to function code */
        default: /* unknown code */
            errno=E_UNKSV;
            break;
    }
    return(errno);
}

/* Set status */
int setstat(f,r)
int f, /* function code */
REGISTERS *r; /* ptr to caller's register stack frame */
{
    register Rbdrive dtb; /* drive table ptr */
    register struct rbf_opt opts; /* path descriptor options ptr */
    dtb=pathdesc->rbfpvt.pd_dtb; /* get drive table ptr */
    opts=&pathdesc->rbfopt; /* point at path descriptor options */
    switch (f) { /* act according to function code */
        case SS_WTrk: /* format track */
            /* Use caller's parameters: */
            errno=format(r->d[2],r->d[3],r->d[4]);
            break;
        default: /* unknown code */
            errno=E_UNKSV;
            break;
    }
    return(errno);
}

```

```

/* Terminate */
int term(dd)
mod_dev *dd;                /* pointer to device descriptor */
{
    /* Remove interrupt handler: */
    f_irq(dd->_mvector,0,NULL,NULL);
    return(0);               /* no error */
}
/* Interrupt service routine */
int irqsvc(port)
void *port;                  /* interface chip address */
{
    return(0);               /* interrupt successfully handled */
}

```

Normally a **make** file would be used to compile and link the source files to make the device driver. As an example, however, typical command lines are shown below:

```

$ r68 rbskel.a -q -o=RELS/rbskel.r
$ cc -qs rbdrv.c -r=RELS
$ l68 RELS/rbskel.r RELS/rbdrv.r -l=/dd/LIB/math.l
-l=/dd/LIB/sys.l -O=OBJS/rbdrv

```

15.13 A FILE MANAGER IN C

The principles described above for writing a device driver in C apply equally well to writing a file manager in C. In addition to receiving calls from the kernel – for which the parameter convention will always be the same – the skeleton must also provide one or more functions to call the device driver routines. The skeleton below has been written with a function (**CallDriver**) to pass a fixed set of parameters to the device driver. This function is suitable for file managers using the same driver calling conventions as used by SCF and RBF, and would also be suitable for any file manager being defined from scratch. Therefore this skeleton can be used without modification for almost any file manager.

The skeleton passes the address of the path descriptor in the **a6** register. This means that any static storage definitions within the file manager source files will be storage within the path descriptor. The file manager writer must take care that the total of such definitions – including the fields used by the kernel – does not exceed the 128 bytes available in the first half of the path descriptor.

The definitions file 'DEFS/path.h' declares a structure describing the path descriptor. It assumes that a file manager definitions file – such as 'DEFS/rbf.h' or 'DEFS/scf.h' – has already been read. To create a new file

manager, the programmer should take a copy of 'DEFS/path.h', and edit it to include the structures declared for his file manager in a new file analogous to 'DEFS/rbf.h'. The "null" file manager shown after the skeleton uses the file 'DEFS/rbf.h', and so has the variables and options section defined by Microware for RBF.

The skeleton shown below includes a stack checking function `_stkcheck()`. The C source files can therefore be compiled with normal stack checking (that is, without the '-s' option).

□ File Manager Skeleton

```
* File 'fmskel.a'
* File manager skeleton
* Non-null psect giving a module type of "file manager":
Typ_Lang      equ      (FIMgr<<8)+0bjct
Att_Revs      equ      ((ReEnt+SupStat)<<8)+0
                psect   fmskel,Typ_Lang,Att_Revs,Edition,0,fmskel
                use     /dd/DEFS/oskdefs.d
*****
* Calling convention from kernel
*
* Passed:   (a1) = Path Descriptor
*           (a4) = Process Descriptor
*           (a5) = Caller's Register Stack Frame
*           (a6) = System Globals
*
* Returns:  carry set if error, with error code in d1.w
*
* Destroys: may destroy ccr, d0-d7/a0-a4
*****
*****
* Calling convention to C routines
*
* The parameters passed to the C routine are:
*   Caller's Register Stack Frame ptr
* The C routine returns:
*   OS-9 error code if error, else 0
*****
*****
* Entry table
*
fmskel
                dc.w    Create-fmskel    create
                dc.w    Open-fmskel      open
                dc.w    MakDir-fmskel     make directory
                dc.w    ChgDir-fmskel     change directory
                dc.w    Delete-fmskel     delete
                dc.w    Seek-fmskel       seek
```

	dc.w	Read-fmskel	read
	dc.w	Write-fmskel	write
	dc.w	ReadLn-fmskel	read line
	dc.w	WriteLn-fmskel	write line
	dc.w	GetStat-fmskel	get status
	dc.w	SetStat-fmskel	set status
	dc.w	Close-fmskel	close
Create:	lea	create(pc),a0	point at C routine
	bra.s	FMCall	
Open:	lea	open(pc),a0	point at C routine
	bra.s	FMCall	
MakDir:	lea	mkdir(pc),a0	point at C routine
	bra.s	FMCall	
ChgDir:	lea	chkdir(pc),a0	point at C routine
	bra.s	FMCall	
Delete:	lea	delete(pc),a0	point at C routine
	bra.s	FMCall	
Seek:	lea	seek(pc),a0	point at C routine
	bra.s	FMCall	
Read:	lea	read(pc),a0	point at C routine
	bra.s	FMCall	
Write:	lea	write(pc),a0	point at C routine
	bra.s	FMCall	
ReadLn:	lea	readln(pc),a0	point at C routine
	bra.s	FMCall	
WriteLn:	lea	writeln(pc),a0	point at C routine
	bra.s	FMCall	
GetStat:	lea	getstat(pc),a0	point at C routine
	bra.s	FMCall	
SetStat:	lea	setstat(pc),a0	point at C routine
	bra.s	FMCall	
Close:	lea	close(pc),a0	point at C routine
	lea	close(pc),a0	point at C routine
* Fall through to FMCall			
* Call the appropriate C function (function address is in a0)			
FMCall	move.l	a6,pd_sysglob(a1)	save System Globals ptr
	movem.l	a5-a6,-(a7)	save registers
	move.l	a5,d0	pass caller's stack frame ptr


```

        movea.l a1,a6          copy Path Desc ptr as C static
        jsr      (a0)          call C routine
        move.l   d0,d1         copy error status
        beq.s    FMCall10      ..no error; leave carry flag clear
        ori      #Carry,ccr    set carry to indicate error
FMCall10  movem.l (a7)+,a5-a6    retrieve registers
        rts

*****
* CallDriver
* C-callable routine to call the device driver
*
* Passed:  d0.l = Offset to routine offset in entry table, e.g. D_READ
*          d1.l = First parameter for driver
*          4(a7) = Second parameter
*          8(a7) = Third parameter
*          12(a7) = Fourth parameter
*          (a6) = Path Descriptor
* Returns: 0 if carry clear from driver, else d1.w extended to int
* Destroys: ccr
*
* Passes:  d0-d3 = Parameters
*          (a1) = Path Descriptor
*          (a2) = Device Static Storage
*          (a4) = Process Descriptor
*          (a5) = Caller's register stack frame
*          (a6) = System Globals
*
* Driver may destroy: ccr, d0-d7/a0-a6
*
CallDriver:
        movem.l d2-d7/a0-a6,-(a7)  save regs
        move.l   d0,d4          copy routine offset offset
        move.l   d1,d0          copy first parameter for driver
        movem.l  56(a7),d1-d3    get three more parameters
        movea.l  a6,a1          copy Path Descriptor ptr
        movea.l  PD_DEV(a1),a2   get Device Table
        movea.l  V$DRIV(a2),a0   get address of driver module
        movea.l  V$STAT(a2),a2   get Device Static Storage ptr
        movea.l  PD_RGS(a1),a5   get caller's stack frame ptr
        movea.l  PD_SysGlob(a1),a6 and System Globals ptr
        movea.l  D_Proc(a6),a4   and Process Descriptor ptr
        add.l    M$Exec(a0),d4   get offset to routine offset table
        move.w   0(a0,d4.l),d4   get offset to routine
        jsr      0(a0,d4.w)      call driver routine
        bcc.s    CallD10        ..no error
        moveq    #0,d0
        move.w   d1,d0          copy error code as a long word
        bra.s    CallD20
CallD10  moveq    #0,d0          indicate no error
CallD20  movem.l (a7)+,d2-d7/a0-a6 retrieve registers
        rts

```

```

*****
* _stkcheck
*   System state stack overflow check, assuming the system state stack
*   is in the process descriptor.
*   Hangs at '_stkerr' if stack overflow
*
*   Passed:  (a6) = Path Descriptor
*            (a7) = stack
*
*   Minimum stack leeway to insist on, in addition to C compiler default:
_stkmin:    equ    200
_stkcheck:  move.l  a0,-(a7)          save register
            movea.l PD_SysGlob(a6),a0 get ptr to System Globals
            movea.l D_Proc(a0),a0    get Process Descriptor ptr
            add.l   a7,d0            calculate desired new stack bottom
            sub.l   #_stkmin+P$Last,d0 with some margin
            cmp.l   a0,d0
            movea.l (a7)+,a0         retrieve register
            bcs.s   _stkerr          ..not enough stack
            rts
_stkerr:
* Insufficient stack - loop forever:
    bra.s  _stkerr
ends

```

Below is the corresponding C source code to make a "null" file manager.

```

/* RBF format file manager in C. Must be linked with 'fmskel.r'.
   Operating system definitions: */
#include <errno.h>          /* error codes */
#include <modes.h>          /* file access modes */
#include <rbf.h>            /* RBF structures */
#include <MACHINE/reg.h>    /* register stack frame */
#include <procid.h>         /* process descriptor */
#include <sg_codes.h>       /* Get/Set status codes */
#include <path.h>           /* common path descriptor structure */

/* Functions in 'fmskel.a':
extern int CallDriver();    /* call the device driver */

/* Static storage definitions (for Path Descriptor): */
union pathdesc pd;         /* path descriptor */
/* Create */
int create(r)
REGISTERS *r;              /* caller's register stack frame */
{
    char *plist;           /* ptr to pathlist */

    plist=r->a[0];         /* get ptr to pathlist */
    return(0);            /* no error */
}

```

```

/* Open */
int open(r)
REGISTERS *r;
{
    char *plist;

    plist=r->a[0];
    return(0);
}
/* Make directory */
int mkdir(r)
REGISTERS *r;
{
    char *plist;

    plist=r->a[0];
    return(0);
}
/* Change directory */
int chgdir(r)
REGISTERS *r;
{
    char *plist;

    plist=r->a[0];
    return(0);
}
/* Delete */
int delete(r)
REGISTERS *r;
{
    char *plist;

    plist=r->a[0];
    return(0);
}
/* Seek */
int seek(r)
REGISTERS *r;
{
    unsigned int pos;

    pos=r->d[1];
    return(0);
}
/* Read */
int read(r)
REGISTERS *r;
{
    unsigned int n;
    unsigned char *p;

```

/* caller's register stack frame */

/* ptr to pathlist */

/* get ptr to pathlist */

/* no error */

/* caller's register stack frame */

/* ptr to pathlist */

/* get ptr to pathlist */

/* no error */

/* caller's register stack frame */

/* ptr to pathlist */

/* get ptr to pathlist */

/* no error */

/* caller's register stack frame */

/* ptr to pathlist */

/* get ptr to pathlist */

/* no error */

/* caller's register stack frame */

/* desired position */

/* get desired position */

/* no error */

/* caller's register stack frame */

/* number of bytes to read */

/* buffer to read to */

```

        n=r->d[1];                /* get number of bytes to read */
        p=r->a[0];                /* get ptr to buffer */
        return(0);               /* no error */
    }
/* Write */
int write(r)
REGISTERS *r;                   /* caller's register stack frame */
{
    unsigned int n;              /* number of bytes to write */
    unsigned char *p;           /* buffer to write from */

    n=r->d[1];                   /* get number of bytes to write */
    p=r->a[0];                   /* get ptr to buffer */
    return(0);                  /* no error */
}
/* Read line */
int readln(r)
REGISTERS *r;                   /* caller's register stack frame */
{
    unsigned int n;              /* number of bytes to read */
    unsigned char *p;           /* buffer to read to */

    n=r->d[1];                   /* get number of bytes to read */
    p=r->a[0];                   /* get ptr to buffer */
    return(0);                  /* no error */
}
/* Write line */
int writeln(r)
REGISTERS *r;                   /* caller's register stack frame */
{
    unsigned int n;              /* number of bytes to write */
    unsigned char *p;           /* buffer to write from */

    n=r->d[1];                   /* get number of bytes to write */
    p=r->a[0];                   /* get ptr to buffer */
    return(0);                  /* no error */
}
/* Get status */
int getstat(r)
REGISTERS *r;                   /* caller's register stack frame */
{
    /* Call the driver's Get Status routine, passing the
       function code: */
    return(CallDriver(D_GSTA,r->d[1]));
}
/* Set status */
int setstat(r)
REGISTERS *r;                   /* caller's register stack frame */
{
    /* Call the driver's Set Status routine, passing the
       function code: */

```

```

    return(CallDriver(D_PSTA,r->d[1]));
}
/* Close */
int close(r)
REGISTERS *r;                /* caller's register stack frame */
{
    return(0);                /* no error */
}

```

Normally a **make** file would be used to compile and link the source files to make the file manager. As an example, however, typical command lines are shown below. Note that the **cc** '-s' option is not used, because a stack checking function is provided in 'fmskel.a':

```

$ r68 fmskel.a -q -o=RELS/fmskel.r
$ cc -q rbmgr.c -r=RELS
$ l68 RELS/fmskel.r RELS/rbmgr.r -l=dd/LIB/math.l
-l=dd/LIB/sys.l -O=OBJS/rbmgr

```

15.14 HINTS ON C PROGRAMMING

C is a very powerful programming language, but it can also be a very confusing language. This section attempts to clarify some of the most common difficulties with C.

15.14.1 Declarations and Definitions

A **definition** is a statement that creates static storage or program code. A **declaration** is a statement that describes an object – a type, or a variable, or a function – without causing the compiler to generate any output. The syntax of definitions and declarations is very similar. For example, a declaration of a function simply omits the arguments and the compound statement body of the function that would make it a definition. A definition of a variable is converted to a declaration by preceding it with the keyword **extern**.

Understanding the syntax of declarations and definitions is one of the most common difficulties in C programming. In fact, once the principle is known, the process is very simple. In short, you should start with the symbol name, and work outwards using the standard order of precedence of the operators, as shown in the book "The C Programming Language", by Kernighan and Ritchie. The example below shows the technique. It is easiest if an appropriate phrase is used to replace each operator. The example shows the declaration of an array (it is a declaration rather than a definition, because the array size is not given), followed by a line for each operator, in the order of precedence, together with an appropriate phrase describing the operator:

```

int *(*fred[])( );
fred                                /* fred is */
fred[]                             /* an array of */
*fred[]                             /* pointers to */
(*fred[])( )                       /* functions returning */
(*(*fred[])( ))                    /* pointers to */
int *(*fred[])( )int               /* int */
/* fred is an array of pointers to functions returning pointers
   to int */

```

Exactly the same technique is used in reverse to create a desired definition or declaration. As with all expressions, parentheses are used to change the order of precedence of operators (as in the example above).

A *cast* operator is exactly the same as placing a declaration of an object of the desired type in parentheses, but omitting the object name. It causes the compiler to convert the value of an object of one type into the form of another type for use in an expression. For example:

```

int (*p)( );                       /* 'p' is a pointer to a function returning
                                   an int */
char *s;                           /* 's' is a pointer to char */
p=(int (*)( ))s;                   /* cast 's' to the type of 'p', and copy it
                                   to 'p' */

```

15.14.2 Pointers and Arrays

The concept of a **pointer** does not appear in most programming languages, but it is essential in a language designed for writing operating system components. In C, a pointer variable contains a memory address, just as a processor address register does. A pointer variable can be assigned any memory address (even an illegal address which will cause a bus error when the pointer is used to access memory). However, a C pointer has an additional property – the type of object it points to. For example, the statement:

```
int *jim;
```

defines a pointer that will be used to address objects of type **int**. Because the type of the objects pointed to is known to the compiler, the compiler can generate correct code when the pointer is used to read or write memory, and when arithmetic operations are used on the pointer. In the Microware 68000 C compiler, an **int** is a 32-bit long word. That is, it requires four bytes of storage. Thus **jim** is used to point to groups of four bytes:

```

int *jim;
int x,y;
jim=(int *)0x00000000;             /* point at memory location zero */
x=*jim;                           /* read the long word at address zero */

```

```

y=jim[7];           /* read the long word at address 28 */
jim=jim+1;          /* add the size of one integer to jim */

```

At the end of this sequence of instructions **jim** has the value 4.

The relationship between pointers and arrays is another very common source of confusion. Again, once the principle is known the confusion disappears. Whereas in most programming languages an array name refers to the whole of the storage used for the array, in C the array name is actually a *constant pointer* to the array storage – it does not refer to the storage itself. Therefore an array name is syntactically a pointer. Its type is a pointer to objects of the type of the array elements. For example:

```
int fred[5];
```

Here **fred** is actually a constant pointer to objects of type **int**. Its "value" is the address of the storage allocated for five integers. To illustrate this, the following statement defines a variable of type pointer to **int**:

```
int *jim;
```

If **jim** is to be used to point to the elements of the array **fred**, the programmer's instinctive reaction is to write:

```
jim=&fred;
```

This is wrong (and syntactically illegal), as it asks the compiler to put into the variable **jim** the address of a constant, **fred**, and of course a constant has no address. In addition, there would be a type mismatch – the address of a pointer to **int** has type pointer to pointer to **int**, which does not match the type of **jim**. Instead, either of the two following statements can be used:

```

jim=fred;
jim=&fred[0];

```

The first statement asks the compiler to put the value of a constant pointer to **int**, **fred**, into the variable **jim**. The second statement asks the compiler to put the address of the first element pointed to by **fred** into the variable **jim**, which evaluates to the same thing (and indeed the Microware C compiler produces the same instructions). Conversely, because **fred** is a pointer to **int**, all pointer operations can be used on **fred**, except operations that attempt to change its value. The statements below show some examples where **fred** and **jim** are used to produce the same results (although the code from the compiler is different). Some illegal statements are also shown, to illustrate the constant nature of **fred**:

```
jim=fred; /* copy the constant value of fred to the variable jim */
```

The next four statements all assign the value 4 to the third element of the array:

```
fred[2]=4;
*(fred+2)=4;
jim[2]=4;
*(jim+2)=4;
```

The following statements illustrate the constant nature of an array name:

```
fred=jim;           /* WRONG - can't assign a value to a constant */
jim++;
fred++;             /* WRONG - can't alter the value of a constant */
```

15.14.3 Pointers and Functions

The use of pointers to functions also often causes confusion. Pointers to functions can be very useful. They allow dynamic configuration of the functionality of a program. Pointers to functions are also useful when a program wishes to use a separate subroutine module – the program links to the subroutine module, and then (using a table of routine offsets at the start of the module) builds an array of pointers to the functions in the module.

A pointer to a function simply contains the memory address of the first instruction of the function. When a pointer to a function is used to call the function, the compiler generates code that loads the pointer into a processor address register, and performs a call to the subroutine at the address in the address register:

```
movea.l funcptr(a6),a0
jsr     (a0)
```

The type of a pointer to a function includes the type of the object returned by the function. For example:

```
double (*mary)();
```

defines **mary** as a pointer to a function returning type **double**. This allows the compiler to make correct use of the value returned by the function when the pointer is used within an expression to call the function.

Just as with arrays and pointers, the type of a function name is exactly the same as the type of a pointer to the function. That is, the function name is a constant pointer to the function. Its value is the absolute memory address of the first instruction of the function. This means that function names and variables that are pointers to functions (returning objects of the same type) can syntactically be used interchangeably. The example below shows the declaration of a function, and the definition of a pointer to a function returning an object of the same type (pointer to **char**).

```
char *fred();      /* function returning a pointer to char */
char *(*jim)();    /* a pointer to such a function */
```


Just as with an array name, it is not possible to take the address of a function name, because it is a constant (a constant pointer to the function code):

```
jim=&fred;      /* WRONG */
```

Instead, to copy the address of the function to a variable (a pointer to a function), the function name alone is used:

```
jim=fred;
```

The constant value **fred** – the address of the function – is copied to the variable **jim**.

Because a function name is a constant pointer to the function, it has all the properties of a pointer, except that it cannot be modified. The example statements below show that once **fred** has been copied to the variable **jim**, the two can be used interchangeably to produce the same results, except that **fred** cannot be modified. In the example statements, **george** is an array of pointers to functions returning objects of type "pointer to **char**":

```
char *fred();    /* declare a function returning a pointer to char */
char *(*jim)();  /* a pointer to such a function */
char *(*george[5])(); /* an array of such pointers */
char *s;        /* 's' is a pointer to char */

jim=fred;        /* copy the address of the function */
george[3]=fred;  /* copy the address of the function */
george[3]=jim;   /* exactly the same result */
s=fred();        /* call the function, and assign the result to 's' */
s=jim();         /* exactly the same effect */
s=george[3]();   /* exactly the same effect */
```

A syntactic laxity in the original specification of C allows an alternative usage of a pointer to a function when calling the function:

```
char *fred();    /* declare function returning a pointer to char */
char *(*jim)();  /* a pointer to such a function */
char *s;        /* 's' is a pointer to char */

jim=fred;        /* copy the address of the function */
s=fred();        /* call the function, assign the result to 's' */
s=jim();         /* exactly the same effect */
s=(*jim)();      /* alternative syntax */
```

The alternative syntax effectively implies that the parentheses operator () operates on the function as a whole – the object pointed to by the pointer. This is clearly not the case – **fred** and **jim** have the same type, as one can be assigned to the other, so if **(*jim)()** is needed, then **(*fred)()** would also be required. Unhappily, it is this alternative syntax that was used in the first edition of Kernighan and Ritchie's book, so it is likely that some C compilers will not accept the first form. The reference section of later editions of K & R

clearly shows that the first form is syntactically the correct form, but also states that in deference to older compilers the ANSI standard allows both forms. The Microware C compiler accepts both forms.

15.14.4 Pointers and Structures

Finally, structures and pointers to structures have often caused confusion. The syntax of C considers a structure to be a data object, exactly the same as a simple data object such as an **int** or a **double**. The structure name refers to the whole of the object. This is very different from an array, where the array name is a constant pointer to the first element of the array. The example below shows the declaration of a structure type, followed by the definition of a structure object of that type, and a pointer to such an object.

```
typedef struct {          /* declare the structure type */
    int item_1;
    char *item_2;
} mystruct;              /* the new type is 'mystruct' */
mystruct fred;           /* 'fred' is an object of that type */
mystruct *jim;           /* 'jim' is a pointer to such an object */
```

The assignment operator '=' will copy the whole of the structure. The structure name refers to the whole of the structure, not the address of the structure:

```
jim=fred;                /* WRONG */
jim=&fred;               /* point at the structure */
```

jim now contains the address of the first byte of the structure. Because the type of the object pointed to by **jim** is known to the compiler, the object pointed to by **jim** can be used in expressions, and simple arithmetic can be used on **jim**:

```
mystruct george;         /* define a new object of that type */
george=fred;             /* copy the structure as a whole */
george=*jim;             /* same effect */
jim=jim+1;               /* add the size of one structure to 'jim' */
```

The elements of the structure can be accessed using both the structure name, and the pointer to the structure. However, the operators used are different:

```
fred.item_1=5;           /* set the first element to the value 5 */
jim->item_1=5;           /* same effect - different operator */
```

The declaration and definition of structures also causes some confusion. A simple structure type declaration has the following syntax:

```
struct henry {           /* declare the structure type */
    int item_1;
    char *item_2;
};
```

This is a declaration of an object type **struct henry** – no storage is reserved at this time. The symbol **henry** is known as the structure tag – it identifies the structure type in subsequent statements. The structure type can be used to define objects:

```
struct henry x;          /* 'x' is a structure */
```

The C language specification allows the two statements to be merged together:

```
struct henry {           /* declare the structure type */
    int item_1;
    char *item_2;
} x,y,z;                 /* 'x', 'y', and 'z' are structures */

struct henry a,b,c;      /* and so are 'a', 'b', and 'c' */
```

The example above merges the declaration of the structure type **struct henry** with the definition of three structures of that type, **x**, **y**, and **z**. The second statement shows that the declared structure tag can still be used to define further objects of that type – **a**, **b**, and **c**. The structure tag can be omitted if only the merged form of the statement is to be used (further structures of that type cannot be defined):

```
struct {
    int item_1;
    char *item_2;
} x,y,z;                 /* 'x', 'y', and 'z' are structures */
```

As with any definition, a structure definition can be converted to the declaration of a new object type by prefixing it with the **typedef** keyword. Thus:

```
struct henry {
    int item_1;
    char *item_2;
} george;
```

declares a structure type **struct henry**, and defines a data object **george** which is a structure. Storage is reserved for **george**. However:

```
typedef struct henry {
    int item_1;
    char *item_2;
} george;
```

declares a new type **george**. No storage is reserved at this time. Objects can be defined with that type (or by using the structure tag, here **henry**):

```
george a,b,c;            /* 'a', 'b', and 'c' are structures */
struct henry x,y,z;      /* 'x', 'y', and 'z' are of the same type */
```

The structure tag can be omitted:

```
typedef struct {
    int item_1;
    char *item_2;
} george;
george a,b,c;          /* 'a', 'b', and 'c' are structures */
```

However, if the structure is to contain pointers to other structures of the same type (for example, in a linked list), the structure tag is necessary, as the compiler will not see the type name until it has reached the end of the type definition, and C is designed to be compiled in a single pass:

```
typedef struct {
    int item_1;
    george *item_2;          /* WRONG */
} george;

typedef struct henry {
    int item_1;
    struct henry *item_2;    /* correct */
} george;
```

Note that in the above example **struct henry** and **george** are the same type, and can be used interchangeably.

APPENDIX A

GLOSSARY

active	Refers to a process that is currently requesting (and may be getting) processor execution time.
application program	A program written to perform a function that is an end in itself, such as a word processor. Contrast with a utility .
assembler	A program to convert assembly language source code to a Relocatable Object File containing machine code instructions.
assembly language	A symbolic (and therefore more easily read and edited) representation of machine code .
battery-backed	Describes a chip or circuit with a battery to maintain its operation when the main power is turned off.
boot	Short for "bootstrap". The bootstrap program in a computer exists in ROM . It is executed when the computer is turned on or reset. Its function is to load the operating system from disk, and execute the cold start function of the operating system. The term "bootstrap" originates from the concept of pulling oneself up by one's bootstraps (boot laces).
bootstrap	See boot .
bug	Programming error.
byte	8 bit binary number or memory location.
chip	An electronic circuit on a small, thin slice of silicon crystal, encapsulated in a plastic or ceramic case,

GLOSSARY

	with metal legs to make electrical contact with a circuit board.
co-processor	A chip , separate from the processor , that adds extra instructions to the instruction set of the processor, by means of a special intimate conversation between the two chips, known as the co-processor protocol. The co-processor may be built onto the same chip as the processor. Examples of this are the MMU in the 68030, and the MMU and FPU in the 68040.
compiler	A program that translates the source code of a high level language program into assembly language .
CPU	Central Processing Unit. See processor .
CRC	Cyclical Redundancy Check. A number of fixed length calculated by passing a data stream through a mathematical expression. A CRC is therefore a value that almost uniquely identifies a given block of data, such as an OS-9 program module , or a sector on a disk. Re-checking the CRC permits the computer to ensure that the data has not been altered or corrupted.
cyclical redundancy check	See CRC .
debugger	A program used to monitor the execution of a program or operating system under test, to look for bugs .
EPROM	Erasable Programmable Read Only Memory. PROM that can be erased with ultra-violet light.
FPU	Floating Point Unit. A co-processor providing floating point arithmetic instructions to supplement the instruction set of the processor . The 68000 and 68010 processors cannot support co-processors. The 68040 has the FPU co-processor built onto the processor chip.
function	See software function .
hexadecimal	The base 16 numbering system.
high level language	A formal programming language that does not

	directly correspond to the instructions the processor can execute. A compiler translates a high level language program into assembly language . C, Pascal, Fortran, Basic, and Modula-2 are all high level languages.
kernel	Under OS-9, the module containing all the essential functions of the operating system, such as process scheduling and memory allocation.
log in	The process of "signing on" to use a computer. Usually initiated by pressing [CR] at a terminal not currently in use.
long word	32 bit binary number or memory location.
machine code	The binary instructions known to (and executable by) the processor .
microprocessor	A processor on a single chip.
MMU	Memory management unit. A circuit (which may be on a single chip , and may be a co-processor) that is placed between the processor and the memory, to convert the processor's addresses into alternative (programmable) addresses, allowing the addressing of the memory to be dynamically altered, usually by blocks of 2 or 4k bytes. The MMU also provides memory protection facilities block by block - for example, a block may be write-protected. The 68030 and 68040 processors have MMU co-processors built into the processor chip.
module	Under OS-9, short for "memory module". A memory module is any block of computer data (including programs and operating system components), prefixed by a descriptive header structure, and terminated by a CRC .
operating system	A collection of software functions for the management of the resources of the computer.
OS-9	"Operating System for the 6809 microprocessor ". Now primarily used on the Motorola 68000 family of microprocessors, and also adapted to 80386 and 80486 IBM PC compatible computers as OS-9000 .
OS-9000	OS-9 adapted to IBM PC compatible computers

GLOSSARY

	and certain other microprocessors .
polling	Repeatedly checking a memory location or device status flag to see if action is required.
portability	A measure of how easy it is to adapt a program that runs on one computer or operating system to run on another computer or operating system.
position independent	Refers to a program or operating system component that executes correctly regardless of the memory address it is loaded at.
process	A program currently running on the computer. A process is not necessarily active – it may be sleeping .
processor	The part of the computer that reads and executes program instructions from memory, and reads and writes data in memory.
program	A list of instructions for the processor to execute to achieve a designed purpose. See application program and utility .
PROM	Programmable Read Only Memory. ROM that can be programmed from a computer using a special device (a PROM programmer).
RAM	Random Access Memory. Computer memory that can be read and written. Unless battery-backed , the contents of the memory are lost when the power is turned off.
re-entrant	Refers to a program or operating system component which does not modify its own instructions, and refers to its data structures relative to a processor address register. This allows the operating system to invoke multiple "incarnations" of the program that use the same program module in memory, by specifying a different data structure address.
relocatable	Under OS-9, the same as position independent . However, under some operating systems (not OS-9) a relocatable program is modified as it is loaded to compensate for the actual load address, so making it position independent.

relocatable object file	See ROF .
ROF	Relocatable Object File. Under OS-9, a file produced by the assembler from an assembly language text file. It contains the machine code instructions, and any public symbols and external references generated by the source code .
ROM	Read Only Memory. Computer memory that cannot be written to without a special programming device, but that retains its contents when the power is turned off.
scheduling	The management of multiple processes concurrently, such that they all get some share of the time of the processor .
semiconductor device	A chip .
sleeping	Refers to a process that is not currently requesting processor time.
software	Processor instructions.
software function	A set of processor instructions to perform a designated operation.
source code	Computer instructions in a textual representation, such as a high level language or assembly language .
systems programming	The writing of operating system components, such as device drivers.
task	Under OS-9, the same as process .
terminal	A device consisting of a keyboard and display, used for interactive input to a computer. A terminal that uses a dynamically modifiable screen for the display (as opposed to using a printer) is sometimes referred to as a VDU .
time slice	A unit of processor time given to each process in turn.
utility	A program designed to facilitate the use of the computer system, such as a utility to copy a disk file.
VDU	Visual display unit – see terminal .

GLOSSARY

word 16 bit binary number or memory location.

APPENDIX B

SBF DEFINITIONS

This appendix lists the assembly language definitions files 'sbfdev.a' and 'sbfpd.a' for the SBF file manager. These files are used by Microware to produce the SBF library file 'LIB/sbf.l'.

□ **sbfdev.a**

```
nam      Sequential Block File Manager
ttl      Static Storage definitions

use      defsfile.d
use      ../../DEFS/iodev.a

psect    sbfdev,0,0,0,0,0

use      ../../DEFS/sbfdev.d

ends
```

SBF DEFINITIONS

□ sbfpd.a

```

        nam      Path Descriptor format
        ttl      Sequential Block File Manager definitions
*****
* Edition History
* #   Date   Changes Made                                     by
* -----
* 1 86/02/13 Created.                                           lac
* 2 88/10/12 Added DMA mode, ScsiID, ScsiLUN.                  wwB
* 3 89/02/27 added scsi options.                               wwB
*
*          <<< ---- OS-9/68000 V2.3 Release ---->>>
*
edition      set      3                      current edition number

        use      defsfile.d

        ttl      Path Descriptor formats
*****
* Path Descriptor Offsets (all file managers)
        org      0
PD_PD:       do.w     1                      Path number
PD_MOD:      do.b     1                      Mode (read/write/update)
PD_CNT:      do.b     1                      number of open images
PD_DEV:      do.l     1                      device table entry address
PD_CPR:      do.w     1                      current process ID
PD_RGS:      do.l     1                      caller's register stack ptr
PD_BUF:      do.l     1                      buffer address
PD_USER:     do.l     1                      User ID of path's creator
PD_Paths:    do.l     1                      linked list of open paths on device
              do.l     4                      reserved
PD_FST:      do.b     128-                  reserve file manager's storage
PD_OPT:      do.b     128                  path options area

        psect    spfpd,0,0,edition,0,0

*****
* Sequential Block Path Descriptor Format

        org      PD_FST
PD_BPTr:     do.l     1                      buffer ptr
PD_BCnt:     do.l     1                      remaining buffer count
PD_DrvTb:    do.l     1                      drive table ptr
PD_DStat:    do.l     1                      driver static storage

        org      PD_OPT
              do.b     1                      device type (all file managers)
PD_TDrv:     do.b     1                      tape drive number
PD_SBF:      do.b     1                      reserved
PD_NumBlk:   do.b     1                      number of blocks (buffers)
PD_BlkSiz:   do.l     1                      maximum block size
PD_Prior:    do.w     1                      driver process priority
```

PD_Flags:	do.w	1	drive capability flags
PD_DMAMode:	do.w	1	DMA type/usage
PD_ScsiID:	do.b	1	controller ID on SCSI bus
PD_ScsiLUN:	do.b	1	tape drive LUN on controller
PD_ScsiOpts:	do.l	1	SCSI option flags
	do.b	256-	reserved

* PD_Flags bit definitions

f_rest_b:	equ	0	rewind on close flag
f_offl_b:	equ	1	offline on close flag
f_eras_b:	equ	2	erase to EOT on close flag

* PD_Flags+1 is free for driver use

ends

Index

/d0

floppy disk drive, 20

/dd

default device, 20, 30

/h0

hard disk drive, 21

/term

terminal, 19

Abort switch

generating level 7 interrupt, 228

Accounting, 241, 367

Activate process

– see '*Signal – activate process*'

Active process

– see '*Process*'

Active queue, 183

Address error, 197, 212

Alarm, 175

absolute, 178

cyclic / periodic, 176

deleting,, 176

relative, 178

single shot, 176

system calls, 241

system state, 178

Alarms

threads, 371

Alias device

– see '*Device – alias*'

Allocation bit map

disk space allocation, 322

Assembler, 82, 87

code segment, 88

conditional assembly, 91

external references, 89, 90

symbol names, 88

Assembly language

called from C, 381

calling C function, 384

Background process, 23, 29

Baud rate

serial communications, 270

Boot, 17

Boot file

installation protected, 77

installing, 75

limitation on size, 76

non-contiguous, 76

Boot ROM, 17

callable functions, 356

Bus error, 197, 212

example handler, 217

in interrupt acknowledge cycle, 227

C

68020/030/040, using, 374

ANSI, 373

assembly language interface, 373,

376

called from assembly language, 384

calling assembly language, 381

cio and math trap handlers, 84, 374

compiler, 81

compiler executive, 81

compiler options, 84

declarations and definitions, 403

external references, 90

FPU, using, 85, 374

libraries, 85

link instruction, 380

Microware compiler, 373

multi-file compilation, 82

optimization, 378

optimizer, 82

pointers and arrays, 404

pointers and functions, 406

pointers and structures, 408

preprocessor, 81

program startup – cstart

– see '*cstart*'

psect generation, 376

register variables, 377

INDEX

- remote keyword, 374
- stack checking, 86, 380
- stack checking in system state, 385
- static storage, large, 374
- static storage addressing, 385
- structure return mechanism, 382
- system call library function, 381
- system state stack usage, 386
- system state system calls, 381
- trap handler written in,, 208
- vsect generation, 385
- Cache memory**
 - coherence (snoopy), 357
 - control, 241, 243
 - disabled during I/O, 110
 - flags, 360
- Catastrophic failure, 241**
- CBOOT, 18, 76**
- CD-I, 11**
- Chain**
 - a process, 24
- Change directory**
 - see *'Directory - change'*
- Character**
 - baud rate
 - see *'Baud rate'*
 - serial, format, 270
- Clock**
 - battery-backed, 20
 - device driver
 - see *'Device driver - clock'*
- Closing**
 - path
 - see *'Path - closing'*
- Cluster**
 - of sectors, 322
 - selecting size, 323
 - size, 322
- Coldstart, 17, 18**
- Coloured memory**
 - see *'Memory - coloured'*
- Command line interpreter**
 - see *'Shell'*
- Common memory**
 - see *'Module - data'*
- Communication**
 - between processes
 - see *'Inter-process communication'*
- Compact Disc Interactive**
 - see *'CD-I'*
- Compatibility**
 - flags, 354, 360
- Compilers, 12**
- Configuration**
 - hardware, 14
- Configuration module**
 - see *'Init'*
- CRC**
 - correct in memory, 239
 - correction, 94
 - generate/check, 47, 238
- cstart, 44, 87, 89, 140, 376**
- Current process, 183**
- Customization, 5**
 - of the kernel, 14
- Cyclical Redundancy Check**
 - see *'CRC'*
- Data available**
 - send signal when,
 - see *'Signal - send, when data available'*
- Data carrier detect**
 - see *'DCD'*
- Data module**
 - see *'Module' and 'Memory'*
- Data rate**
 - serial communications
 - see *'Baud rate'*
- Date**
 - change, 238
 - changed, 178
 - convert to Gregorian, 241, 248
 - convert to Julian, 239
 - get current,, 238
 - Gregorian
 - see *'Gregorian date'*

Julian
 – *see* 'Julian date'

DCD
 send signal when asserted, 290
 send signal when negated, 290

Deadlock, 363

Debugger, 82
 arithmetic expressions, 94
 breakpoints in ROM, 95
 C source level, 82, 95
 cio.stb not found, 94
 execute process, 239
 fork process, 239
 full speed execution, 95
 invoke ROM-based, 241, 254
 modifying modules with,, 95
 program, assembly level, 93
 relocation registers, 94
 ROM-based, 18, 220, 360
 system state, 82
 terminate process, 239
 traced execution, 95
 use of symbol modules, 91, 94

Default device
 making device descriptor, 133

Definitions
 C
 – *see* 'C – declarations and definitions'
 symbolic
 – *see* 'Symbolic definitions'

Definitions files, 341

Delete file
 – *see* 'File – delete'

DevCon, 131

Device
 access interlock, 349
 alias, 30, 98, 260
 as a file, 324
 attaching, 99, 111
 attributes, 108
 concurrent access to,, 314, 335
 descriptor, 13, 97, 103

descriptor, creating, 129
 descriptor for default device, 133
 descriptor for RBF, 132
 descriptor for SBF, 134
 descriptor for SCF, 130
 descriptor header, 103
 descriptor options, 98, 103, 122
 detaching, 100, 113
 initialization, 99, 111
 multi-channel, 335
 names, 21
 permissions, 108
 physical,, 98
 sharable, 108, 115
 table, 98, 259, 348
 termination, 100, 113
 termination, preventing,, 100
 type code, 121
 use count, 101, 113, 117, 325, 348

Device control
 – *see* 'Path descriptor – options', and
 'Device descriptor – options'

Device driver, 9, 13, 97, 102, 257
 C, written in, 384, 387
 clock, 308
 creating, 261
 custom functions, 103, 284
 example skeleton, 305
 get status, 284
 initialization, 100, 274
 kernel calls to,, 102
 multi-channel, 260
 pipes, 102
 psect, 274
 read, 277
 routines, 273
 set status, 284
 termination, 100, 276
 write, 281

Device static storage, 98, 258, 348

Device table
 pointer in path descriptor, 268

Direct command

INDEX

to device, 293

Directory, 106

change, 24, 101, 117
create, 101, 114, 116, 319, 325
current, 20, 27, 73, 117, 319, 324, 364
data, 27, 107, 117
deleting a,, 327
entry, 320
execution, 20, 27, 107, 117
hierarchy, 26, 106
initial current,, 73, 74
initial size, 326
make
– see '*Directory – create*'
opening, 116
root, 106, 319, 322
searching, 26
structure, typical, 30
time last modified, 325
tree structured, 319

Disk

as a file
– see '*Device – as a file*'
cluster size, 75
configuration, 123
filing
– see '*RBF*'
format protected, 77, 267
formatting, 74
identification sector, 76, 322
sector zero, 76, 322
space allocation, 322

Disk caching, 134

Disk capacity

automatic detection, 289

Disk files, 72, 164

attributes, 72
checking structure, 72
owner ID, 72
segment, 267
size limitation, 75

Disk format, 322

changing, 266

Microware codes, 266

Disk transfer

buffer address, 268
device limit, 268
multi-sector, 267

Divide by zero, 197, 214

Drive table, 260

pointer in path descriptor, 268
read from sector zero, 280

Echo

terminal, 268, 318

Editor

– see '*Umacs*'

End of file

character, 269

End of record

character, 269, 330

Environment variables, 24, 74

Error

abort on, 25
convention, 273
messages, 24
numbers, 24
symbolic names, 272

Ethernet, 136

Event, 149

auto increment, 150
change value, 151
comparison with signal, 149
creating an,, 150
deleting, 151
examples of usage, 154
group wakeup, 151
ID, 352
index, 352
link count, 151
linking to,, 151
number, 352, 357
pulsing, 152
read current value, 151
signal received while waiting
– see '*Signal – while waiting for*'

- event'*
- sub-code, 153
- system call, 241
- table, 351
- value, 150
- wait for., 148, 150, 152
- wakeup if deleted, 151
- wakeup increment, 150

Exception

- handling, 197, 238
- hardware, 197, 212
- hardware, user state handler, 213
- in interrupt handler, 220
- in system state, 202, 220, 355
- jump table, 198, 230
- numbers, symbolic, 213
- program exit status, 212
- return from., 199
- stack frame, 214
- system state handler, 220
- trap #n instruction, 202
- vector table, 198, 229

Execute

- mode
- *see 'Path – execute mode'*

Exit process

- *see 'Process – termination'*

File, 106

- attributes, 108
- create, 109, 114, 318
- date created, 321
- delete, 101, 117
- directory, 106
- disk, 106
- *see 'Disk files'*
- end of, character
- *see 'End of file – character'*
- extending, 323
- initial size, 109, 320
- mode
- *see 'Path – mode'*
- name
- *see 'Pathlist'*

- names, valid., 107
- open, 318
- owner, 110, 321
- permissions, 108, 110, 319
- pointer, changing, 117
- read
- *see 'Path – read'*
- segmentation, 320
- time last modified, 321
- undelete, 327
- write
- *see 'Path – write'*

File descriptor, 75, 320

- caching, 320

File manager, 9, 12, 97, 102, 313

- C, written in, 384, 396
- concurrent access to device, 314, 335
- custom functions, 331
- data editing, 329
- device driver, comparison with, 313
- device driver calls, 333
- example skeleton, 336
- I/O queueing in., 335
- I/O unqueueing, 336
- kernel interface to., 316
- multi-channel, 335
- path descriptor usage, 317
- pathlist parsing, 318
- type code, 121

File manager functions, 315

- change directory, 324
- close, 333
- create, 318
- delete, 327
- get status, 331
- make directory, 325
- open, 318
- read, 328
- read line, 329
- seek, 328
- set status, 331
- write, 328

INDEX

- write line, 329
- File names**
 - extensions, 83
 - wild cards in,, 22, 107
- Floating point unit**
 - see 'FPU'
- Fork**
 - parameter string, 140
 - process
 - see 'Process - creating'
- Format**
 - disk, 74, 291
 - utility, 323
- Format codes**
 - for disks
 - see 'Disk format - Microware codes'
- FPU**
 - context saving, 200, 367
 - current user, 355
 - in interrupt handler, 226
 - presence, 354
- Gregorian date, 249**
- Group number, 71**
- Handshake**
 - hardware
 - see 'DCD', 'RTS'
 - software
 - see 'XON/XOFF'
- I/O, 97**
 - custom functions, 119, 120
 - device independent,, 9, 110
 - dynamic configuration, 15, 99, 271
 - initialization
 - see 'Device'
 - interface, 98
 - interlock, 239
 - see 'I/O - queueing'
 - modules not automatically loaded, 112
 - reschedule after,, 111, 116, 316
 - sub-system, 98
 - system calls, 110
 - tree-structured
 - see 'Tree-structured I/O'
 - unified, 9, 110
 - unlink module, 240, 250
 - wild card calls, 110, 119, 120
- I/O queueing, 115, 239, 314, 349, 366**
- Illegal instruction, 197, 212**
- Init, 19, 52, 73, 235**
- Initialization of program data**
 - see 'Program - initialization of data'
- Instruction emulation**
 - move from ccr, 219
 - move from sr, 219
- Inter-process communication, 4, 137, 138**
- Inter-process synchronization, 138**
- Internet, 136**
- Interrupt polling table, 351**
- Interrupts, 293**
 - see 'Exception'
 - 68000 mechanism, 222
 - acknowledge cycle, 297
 - auto-vectored, 223
 - bus error in acknowledge cycle, 227
 - clock tick, 310
 - communication with process, 350
 - device driver, wakeup, 298
 - device drivers, in, 293
 - handler, installing, 224, 239, 294
 - handler, removing, 296
 - handlers, multiple on same vector, 296
 - handling in program, 60
 - in multi-tasking applications, 137, 293
 - invalid vector, 227
 - level, choosing, 304
 - level 7, 222, 228
 - mask lowered in handler, 296
 - masked during system calls, 139

- masking, 222, 301
- missed, avoiding,, 299, 301
- necessary operations, 297
- nested, 296
- non-maskable
- see *'Interrupts – level 7'*
- normal vectored, 223
- not required, 293
- OS-9 cannot handle, 227
- polling table, 224
- prioritized, 222
- priority, software polling, 224, 294
- registers preserved, 226, 295
- return from,, 225
- serial port, transmit, 303
- serial port example, 301
- signals, distinct from, 297
- signals, used with, 299
- solicited, 293, 298
- spurious, 227
- stack used during,, 354, 355
- see *'Stack – in interrupt handler'*
- static storage, 295
- system calls while handling,, 296
- unknown origin, 225, 354
- unsolicited, 293, 301
- used in OS-9, 223
- vector, only one device on, 294, 297
- vectors, 294
- Julian date, 248, 354**
- Jump table, 84, 92**
 - exception,
 - see *'Exception – jump table'*
- Kernel, 8, 12**
- Key**
 - abort, 29, 269
 - interrupt, 29, 269
 - quit, 29, 269
- Kill a process, 24**
 - see *'Signal – kill process'*
- Last process to use device, 349**
- Libraries, 85, 90, 91**
- Line editing, 28, 331**
 - buffer, 319
- Line feed**
 - automatic, 268, 330
- Link**
 - to module, 237
- Linker, 82, 91**
 - libraries with,, 91
 - linkage map, 92
 - not OS-9 output, 93
 - options, 92
- Load**
 - modules from file, 237, 250
 - the operating system, 17
- Log in, 20, 73, 74**
- Log out, 24**
- Make utility, 82**
 - file time resolution, 321
- Memory**
 - access to protected,, 57, 174, 240, 252
 - address translation, 50, 55
 - allocate by colour, 241
 - allocation, 50, 53, 239
 - allocation for program, 45, 51
 - allocation priority, 51, 53
 - attributes, 52
 - battery-backed, 52
 - check access permission, 241, 245
 - coloured, 51, 173
 - coloured, data module in,, 245
 - copy, 238, 240
 - create module in coloured,, 52
 - de-allocation, 239
 - deny access permission to,, 240, 253
 - direct access to,, 174
 - external, 173
 - finding during coldstart, 52
 - fragmentation, 51
 - free currently, 51
 - general system,, 52
 - get copy of free list, 238, 247
 - hidden, 52
 - load module into coloured,, 52, 250

- named, 169
- on CPU board, 52
- process minimum block size, 56, 356
- protection, 55
- read only, 52
- system minimum block size, 56, 356
- type, 52
- uncoloured, 52
- with special properties, 52
- Memory address**
 - translate, 242
- Memory management**
 - hardware, 50, 55
 - no need for, 15
- Memory Management Unit**
 - see 'MMU'
- Memory map**
 - get copy of process's,, 241, 249
- Memory structures**
 - OS-9,, 341
- Microware, 11**
- MMU, 55**
- Modular**
 - operating system, 9
- Module, 9, 14, 33**
 - attributes, 40
 - CRC
 - see 'CRC – generate/check'
 - create data,, 239, 245
 - data,, 56, 169
 - data, in coloured memory, 245
 - default trap entry point, 44
 - edition number, 43
 - execution offset, 89
 - group, 48, 50
 - header parity, 43
 - header structure, 33
 - held in memory, 34
 - in file, 49
 - in ROM, 34, 50
 - in the boot file, 50

- install in directory, 239
- language, 39
- link count, 48
- linking to a,, 56
- load into coloured memory, 250
- loading, 34, 48
- minimum data space, 45
- minimum stack size, 45
- name, 33
- offsets in, 35
- permissions, 37, 93
- program entry point, 44
- program in, 35
- replacing in memory, 42
- revision number, 40, 42
- sharable, 40
- sticky / ghost, 34, 40, 93
- supervisor / system state, 40
- sync word, 35
- type, 35, 39
- unlinking, 50
- unlinking I/O,, 50
- user number, 37
- Module directory, 33, 347**
 - find entry, 241
 - get copy of,, 238
- Move from ccr, 219**
- Move from sr, 219**
- Multi-channel device**
 - see 'Device – multi-channel'
- Multi-sector disk transfers**
 - see 'Disk transfer – multi-sector'
- Multi-tasking, 3, 6, 57, 183**
 - applications, 4
 - inter-process communication
 - see 'Inter-process communication'
 - interrupts used for,, 293
- Multi-user, 3**
- Named memory**
 - see 'Memory – named'
- Non-maskable interrupt**
 - see 'Interrupts – level 7'
- OS-9000, 10**

os9 macro, 201, 234

Overflow trap, 197, 212

Parameters

to program

– see *'Program – parameters to'*

Parity

serial

– see *'Character – serial, format'*

Passwords, 71, 72

Path, 105

closed automatically, 106

closing, 120

device linked list of,, 350

duplication, 113, 161, 345

execute mode, 107

get status, 119

get status by system path number,
120

inheriting, 114, 346

link to device, 348

mode, 107, 108

opening, 107, 108, 111, 115

read, 118

read editing, 118

read line, 118

set status, 120

sharable, 115

use count, 113

write, 118

write editing, 118

write line, 118

Path control

– see *'Path descriptor – options'*

Path descriptor, 106, 345

file manager usage, 101

find, 239

initialization, 115

linked list of,, 115

options, 121, 263, 345

options, changing, 122

RBF

– see *'RBF – path descriptor'*

SCF

– see *'SCF – path descriptor'*

table, 345, 350

use by device driver, 263

Path number, 105

local,, 111, 345

system,, 111, 121, 345, 350, 364

Pathlist, 26, 106

name comparison, 238

name validation, 107, 238

wild cards in,, 107

Pause

end of line, 30, 269, 330

end of page, 268, 329, 330

Pipe, 159

buffer size, 160

connecting processes, 162

full when writing, 163

in command line, 23

named, 159, 163

process synchronization, 160

redirection, 162, 163

unnamed, 159, 161

Port address, 348

Portable OS-9

– see *'OS-9000'*

Position independent, 88

Printer

configuration, 126

Process, 57

activating, 186

activating, high priority, 187

active, 58, 183

age, 362

concurrent execution, 59

condemned, 199

creating, 140, 237

current

– see *'Current process'*

dead, 58, 60

disinherited, 60

exit status, 60, 141

high priority, 194

ID, 57, 59, 238

INDEX

- make active, 239
 - make current, 239
 - maximum age
 - see '*Scheduling – maximum age*'
 - memory allocation, 56
 - minimum priority
 - see '*Scheduling – minimum priority*'
 - none to run, 185
 - parent, 60
 - priority, 24, 57, 73, 186, 238
 - priority, changing, 195
 - queues, 356
 - sleeping, 58
 - state flags, 362, 370
 - switching
 - see '*Task switching*'
 - system state, 60
 - termination, 59, 141, 237
 - termination by signal
 - see '*Signal – kill process*'
 - user ID, 238
 - waiting, 58
- Process descriptor, 57, 345, 361**
- get address of,, 248
 - get copy of,, 238
 - get copy of table, 239
 - size, 357
 - table, 59, 350
- Process ID, 350**
- Processor**
- type in use, 356
- Professional OS-9, 12**
- Program**
- initialization of data, 45
 - parameters to, 45
 - static storage, 45, 89
- Program module**
- see '*Module – program in*'
- Psect**
- non-root, 89
 - root, 44, 87, 88
 - syntax, 88
- RAM disk, 166**
- as default device, 31
 - prevent termination, 100
 - termination, 101
- RBF, 26, 97, 314**
- allocation bit map, 322
 - options, 123
 - path descriptor, 263
 - robust filing structure, 322
- Re-entrant, 15**
- Real time, 6**
- applications, 137, 183
 - interrupts used for,, 293
 - kernel, 2
- Record locking, 164**
- deadlock, 165
- Redirection, 9, 65, 114**
- in command line, 23
 - to pipe
 - see '*Pipe – redirection*'
- Register**
- saving by kernel, 273
 - usage in device driver, 272
- Relocatable**
- Object File
 - see '*ROF*'
 - operating system, 14
 - program, 14, 46
- Reset, 197**
- Reset stack pointer, 342**
- Resource management, 5, 341**
- Restore**
- disk drive head, 291
- ROF, 34, 83, 87, 90, 91**
- in library, 91
- ROM**
- operating system in, 8
 - programs in, 4, 8
- ROM disk, 167**
- ROMbug**
- see '*Debugger – ROM-based*'
- Root**
- directory

- see *'Directory - root'*
- psect
- see *'Psect - root'*
- Round robin, 183**
- RTS**
 - assert, 291
 - negate, 291
- SBF**
 - options, 128
- SCF, 28, 97, 314**
 - concurrent write, 30
 - line editing, 331
 - options, 125
 - path descriptor, 268
- Scheduling, 6, 183**
 - after I/O operation, 111, 316
 - automatic, 188
 - constant, 187, 190, 192
 - hierarchical, 192
 - high priority processes, 195
 - in system state, 195
 - manual control, 193
 - maximum age, 192
 - minimum priority, 186, 190
 - options, 185
 - pre-emption, 186, 190, 192, 193
 - pre-emption, rarely needed, 195
 - pre-emption on wakeup, 195
 - pre-emptive, 186
 - precedence of mechanisms, 194
 - process priority, 188
 - real time applications, 194
 - round robin, 186, 188
 - round robin with hierarchical, 192
 - seizing control, 193
 - suspended during interrupt service, 196
 - suspension, 195
 - system age, 187
 - time slicing, 188
- SCSI**
 - device driver, 133, 135, 260
 - direct command, 293

- Sector size**
 - variable, 289
- Seek**
 - within file, 117
- Segment**
 - disk file
 - see *'Disk files - segment'*
- Serial port**
 - dynamic configuration, 271
- Shared memory**
 - see *'Module - data', and 'Memory - external'*
- Shell, 20, 21**
 - prompt, 24, 74
 - signals to, 29
 - special characters, 22
- Signal, 142**
 - activate process, 143
 - after time delay
 - see *'Alarm'*
 - asynchronous response to,, 143
 - broadcast, 142
 - cancel request, 291
 - deadly, 144, 302
 - distinct from interrupt, 142
 - exit handler, 239
 - handler, calling,, 199
 - handler routine, 144, 238
 - kill process, 144, 145
 - mask, 241
 - masked during handling, 145
 - masking, 145
 - most recent, 363
 - periodic
 - see *'Alarm'*
 - received in system state, 143, 147
 - received in user state, 147
 - received while masked, 145
 - send, 237
 - send, when data available, 290
 - terminating receiver, 144
 - wakeup, 145, 148
 - while sleeping, 146

INDEX

while waiting for child, 146
while waiting for event, 148, 153

Sleep, 238

Spurious interrupt

– see *'Interrupts – spurious'*

**SSM, 55, 170, 174, 240, 241, 246,
249, 252, 253**

presence, 357

static storage, 357

task switch, 200

Stack

in interrupt handler, 226

size for program, 92

system state, 201, 349, 361

user state, 362

Standard paths, 114

Startup file, 20, 73

Static storage of program

– see *'Program – static storage'*

Super user, 71

Symbol table modules, 84

Symbolic definitions

for device driver, 271

Sysgo, 19

System age

– see *'Scheduling – system age'*

System build, 35

System calls, 233

called from C, 235

customized, 235, 240

description, 236

errors returned by,, 234

indivisible, 191, 195

install new,, 240

parameters to,, 234

routine address table, 233, 352

system state, 233, 236

trap #0 instruction, 198, 200

user state, 233, 236

System globals, 342, 353

read / modify, 239, 353

user space, 199, 359

System information

get copy of,, 241, 255

System manager, 71

System process, 19, 177, 355

System Security Module

– see *'SSM'*

System shutdown, 241

System state stack

– see *'Stack – system state'*

System tables

dynamically extendible, 342

systype.d

device descriptor macros, 129

Tabs

expansion of,, 330

Tape drive

configuration, 128

Task number, 362

Task switching, 184

TCP/IP, 136

Terminal, 19

concurrent write to

– see *'SCF – concurrent write'*

configuration, 125

Terminate process

– see *'Process – termination'*

Thread

– see *'Alarms – threads'*

Tick, 6, 58

device driver

– see *'Device driver – clock'*

Tick handler, 353

calling custom routine

– see *'Alarm'*

sending signal

– see *'Alarm'*

Ticks

per second, 309

per time slice, 185

remaining in second, 358

Time

change, 238

changed, 178

get current,, 238

seconds until midnight, 354
 since startup, 355
 ticks remaining in second, 358
Time slice, 3, 58, 184
 ticks per,, 185, 358
 ticks remaining in,, 358
Trap handler, 13, 198
 C, written in, 208
 called from system state, 204
 calling, 203
 cio, 84, 203, 374
 initialization, 206
 installing, 203, 206
 link to, 239
 math, 86, 203, 374
 program default entry point, 44
 routines, 204
 system state, 203
 termination, 208
 trap #n instruction, 202
Tree-structured I/O, 13, 15, 97, 135
Tsmon, 20, 74
Umacs, 12, 83
Undelete a file
 - *see 'File - undelete'*
UNIX, 6
 similarity to, 16
Unlink
 from module, 237
 I/O module, 240, 250
User ID, 71, 73
User state
 return to,, 199
Utilities
 formal syntax notation, 65
 general purpose, 68
 operating system functions, 66
 syntax, 64
 system management, 68
Vector base register, 229
Vector table
 exception,

- *see 'Exception - vector table'*
Verify
 after disk write, 267
Vsect, 89
Wait
 for child to die, 24, 60, 140, 141,
 148, 237
Whole device
 as a file
 - *see 'Device - as a file'*
Wild cards
 in file names, 22
Windows
 graphical user interface
 - *see 'X Window System'*
X Window System, 136
XON/XOFF
 flow control, 269



Paul Dayan - the author

The OS-9 Guru - 1 : The Facts is an introductory and technical reference book about the OS-9 operating system. OS-9 is the industry standard real time operating system for the Motorola 68000 family of microprocessors, and - under the name CD-RTOS - is the operating system in all CD-I players. Intended as a companion to the OS-9 User's and Technical Manuals, this book answers many of the questions asked by new users of OS-9, and provides detailed information about the internal workings of the operating system for the benefit of experienced users.

"Before reading **The OS-9 Guru**, I had been working for more than ten years with OS-9/6809 and OS-9/68000. So I knew 'all' about OS-9. After reading the book, I finally understand the operating system."

Ole Hansen, Danelec Electronics

"This book provides a wealth of information which has not previously been readily available, together with many valuable ideas for the effective use of OS-9. I recommend it."

Tony Mountifield, Microware Systems (UK)

"**The OS-9 Guru** gives a good all-round coverage of OS-9 and how to use it, whether one is a beginner or an experienced user."

Nick Rainey, Microware Systems (France)

"**The OS-9 Guru** clearly explains the how, where, what, why, and when of OS-9. Packed with detail, it is essential programmer reading material."

Steve Weller, Windsor Systems

A Galactic Industrial publication

ISBN 0 9519228 0 7



THE OS-9 GURU 1 - The Facts

Paul S. Dayan

Galactic